Design of the Programming Language Forsythe

John C. Reynolds

June 28, 1996

CMU–CS–96–146

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To be reprinted in **Algol-like Languages**, edited by P. O'Hearn and R. D. Tennent, Birkhauser, 1996

Abstract

This is a description of the programming language Forsythe, which is a descendant of Algol 60 intended to be as uniform and general as possible, while retaining the basic character of its progenitor.

This document supercedes Report CMU–CS–88–159, "Preliminary Design of the Programming Language Forsythe" [1].

Research suuported by National Science Foundation Grant CCR-9409997.

Keywords: Forsythe, Algol-like languages, Algol 60, intersection types

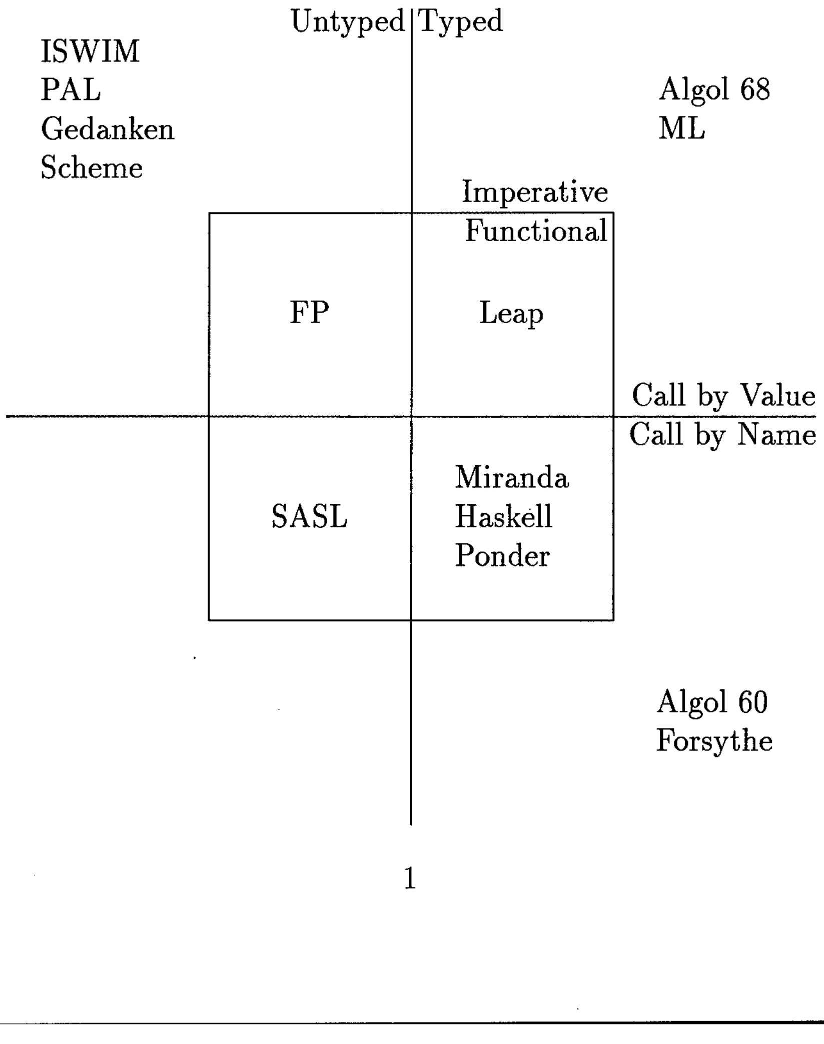
1. Introduction

In retrospect, it is clear that Algol 60 [2, 3] was an heroic and surprisingly successful attempt to design a programming language from first principles. Its creation gave a formidable impetus to the development and use of theory in language design and implementation, which has borne rich fruit in the intervening thirty-six years. Most of this work has led to languages that are quite different than Algol 60, but there has been a continuing thread of concern with languages that retain the essential character of the original language [4, 5]. We feel that research in this direction has reached the point where it is desirable to design a modern Algol-like language that is as uniform and general as possible.

This is the goal of the programming language Forsythe. We believe that it retains the essence of Algol 60, yet is both simpler and more general. The key to achieving this combination of simplicity and generality is to exploit the procedure mechanism and the type system, in order to replace a multitude of specialized features by a few general constructions.

The language is named after George E. Forsythe, founding chairman of the Computer Science Department at Stanford University. Among his many accomplishments, he played a major role in familiarizing American computer scientists (including the author) with Algol.

Before considering Forsythe in detail, we specify its location in the design space of programming languages. As illustrated below, Forsythe lies on one side of each of three fundamental dichotomies in language design:



First, it is a typed language. It has long been understood that imposing a type discipline can yield major improvements in compile-time error detection and in the efficiency of run-time data representations. However, type systems that are flexible enough to support sophisticated programming techniques are a much more recent development.

Second, Forsythe has imperative features (i.e. assignment and control flow) as well as a powerful procedure mechanism. Like all such languages, it suffers from the problems of aliasing and interference. However, we believe that imperative programming is a fundamental paradigm that should not be ignored in programming language design.

Finally, Forsythe uses call by name rather than call by value. For purely functional languages this is merely a distinction between orders of evaluation, but for languages with imperative features it is a fundamental dichotomy in the way that the imperative and functional aspects are linked; one is tempted to speak of Algol-like versus ISWIM-like languages.

In any event, the following basic operational view, which is implicit in Algol 60, underlies Forsythe and distinguishes it from such languages as ISWIM [6], Algol 68 [7], Scheme [8], and ML [9]: The programming language is a typed lambda calculus with a primitive type **comm**(and), such that terms of this type, when reduced to normal form, are commands in the simple imperative language. Thus a program, which must be a term of type **comm**, is executed in two phases. First the program is reduced to normal form. (In Algol jargon, the copy rule is repeatedly applied to eliminate procedure calls.) Then the resulting simple imperative program is executed:

$$\begin{tikzcd}
\text{Reduction of lambda expressions (copy rule)} \arrow[d, "\text{normal form}"] \\
\text{Execution of commands in the simple imperative language}
\end{tikzcd}$$

The only complication is that, in the presence of recursion, the reduction phase may go on forever, producing an infinite or partial "normal form". Nevertheless, such an infinite term can still be viewed as a simple imperative program; operationally, one simply implements the two phases as coroutines.

Even in this more general situation, the above diagram still describes an essential restriction on the flow of information: Nothing that happens in the second phase ever affects anything that happens in the first phase. Thus Forsythe inherits the basic property of the lambda calculus that meaning does not depend upon the order or timing of reductions. Indeed, reduction rules can be viewed as equations satisfied by the language.

In contrast, consider the situation in an ISWIM-like language such as Scheme or ML that provides assignable function variables. If f is such a variable, then the effect of reducing $f(\cdots)$ will depend upon when the reduction occurs relative to the sequence of assignments to f that are executed in the imperative phase. In this situation, the procedure mechanism is completely stripped of its functional character.

2. From Algol to Forsythe: An Evolution of Types

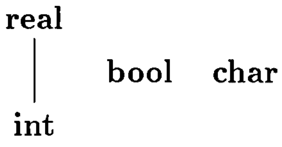
The long evolution which has led from Algol 60 to Forsythe is too complex to recount here in detail. However, to provide an overview of Forsythe and reveal its relationship to Algol, it is useful to outline the development of the heart of the language, which is its type structure. (In this introductory account, we retain the familiar notations of Algol, rather than using the novel notations of Forsythe.)

An essential characteristic of an Algol-like language is that the variety of entities that can be the value of a variable or expression is different from the variety of entities that can be the meaning of identifiers or phrases. We capture this characteristic by distinguishing two kinds of type (as in [5] and [10]):

- A *data type* denotes a set of values appropriate to a variable or expression.
- A *phrase type*, or more simply a *type*, denotes a set of meanings appropriate to an identifier or phrase.

In Algol 60, there are three data types: **integer**, **real**, and **boolean**. In Forsythe, we use more succinct names, **int**, **real**, and **bool**, and add a fourth data type, **char**, denoting the set of machine-representable characters.

To capture the existence of an implicit conversion from integers to reals, we define a partial order on data types called the *subtype* relation. We write $\delta \leq \delta'$, and say that δ is a *subtype* of δ' when either $\delta = \delta'$ or $\delta =$ **int** and $\delta' =$ **real**, i.e.



In Algol 60, the phrase types are the entities, such as **integer**, **real array**, and **procedure**, that are used to specify procedure parameters. However, the phrase types of Algol 60 are not sufficiently refined to permit a compiler to detect all type errors. For example, in both

$$\textbf{procedure}\ \mathit{silly}(x);\ \textbf{integer}\ x;\ y := x + 1$$

and

$$\textbf{procedure}\ \mathit{strange}(x);\ \textbf{integer}\ x;\ x := x + 1$$

the formal parameter x is given the type **integer**, despite the fact that an actual parameter for *silly* can be any integer expression, since x is evaluated but never assigned to, while an actual parameter for *strange* must be an integer variable, since x is assigned to as well as evaluated.

To remedy this defect, one must distinguish the phrase types **int**(eger) **exp**(ression) and **int**(eger) **var**(iable), writing

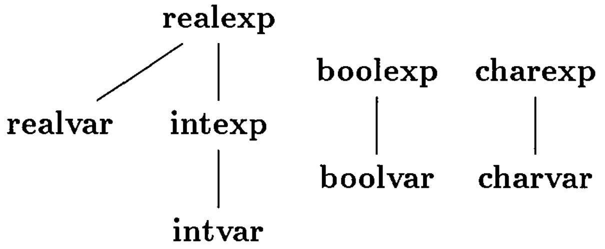
procedure *silly*(x); **intexp** x; $y := x + 1$

and

procedure *strange*(x); **intvar** x; $x := x + 1$.

(In a similar manner, each of the other data types gives rise to both a phrase type of expressions and a phrase type of variables.)

Like data types, phrase types possess a subtype relationship. Semantically, $\theta \leq \theta'$ means that there is an implicit conversion from meanings of type θ to meanings of type θ'. But the subtype relation can also be interpreted syntactically: $\theta \leq \theta'$ means that a phrase of type θ can be used in any context requiring a phrase of type θ'. Thus, since a variable can be used as an expression, **intvar** $\leq$ **intexp**, and similarly for the other data types. Moreover, since an integer expression can be used as a real expression, **intexp** $\leq$ **realexp**. In summary:

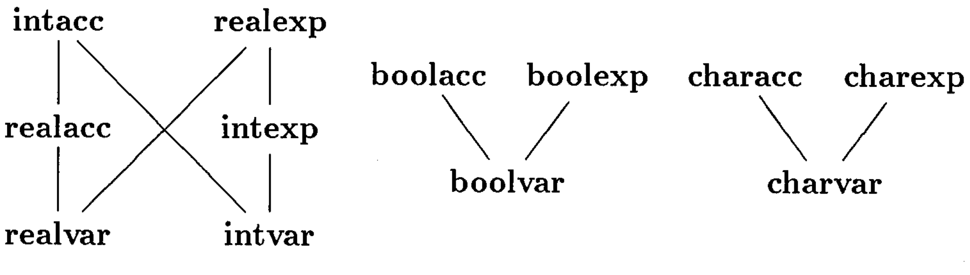


However, there is an unpleasant asymmetry here. It can be remedied by distinguishing, in addition to expressions which can be evaluated but not assigned to, *acceptors* which can be assigned to but not evaluated. Then, for example, we can write

procedure *peculiar*(x); **intacc** x; $x := 0$

to indicate that *peculiar* assigns to its parameter but never evaluates it.

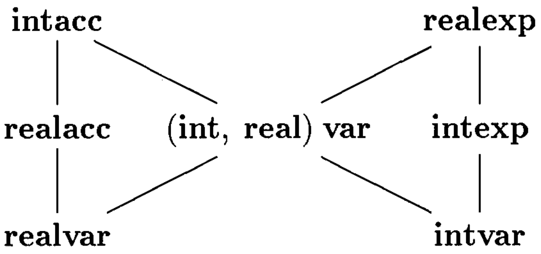
Clearly, **intvar** $\leq$ **intacc**, and similarly for the other data types. Moreover, **realacc** $\leq$ **intacc**, since an acceptor that can accept any real number can accept any integer. Thus the subtype relation is



However, there is a further problem. In Forsythe, the conditional construction is generalized from expressions and commands to arbitrary phrase types; in particular one can construct conditional variables. Thus if p is a boolean expression, n is an integer variable, and x is a real variable, one can write

$$\textbf{if } p \textbf{ then } n \textbf{ else } x$$

on either side of an assignment command. But when this construction occurs on the right of an assignment, it must be regarded as a real expression, since p might be false, while when it occurs on the left of an assignment, it must be regarded as an integer acceptor, since p might be true. Thus the construction is an **int**(eger accepting), **real** (producing) **var**(iable), which fits into the subtype relation as follows:



Next, we consider the types of procedures. In Algol 60, when a parameter is a procedure, one simply specifies **procedure** for a proper procedure (whose call is a command), or **integer procedure**, **real procedure**, or **boolean procedure** for a function procedure (whose call is an expression). But to obtain full compile-time typechecking, one must use more refined phrase types that indicate the number and type of parameters, e.g.

$$\textbf{procedure(intexp, intvar)}$$

to denote a proper procedure accepting an integer expression and an integer variable, or

$$\textbf{real procedure(realexp)}$$

to denote a real procedure accepting a real expression. (Note that this refinement introduces an infinite number of phrase types.)

These constructions can be simplified and generalized by introducing a binary type constructor $\rightarrow$ such that $\theta \rightarrow \theta'$ denotes the type of procedures that accept θ and produce θ' or, more precisely, the type of procedures that accept a single parameter of type θ and whose calls are phrases of type θ'. For example, a real procedure accepting a real expression would have type $\textbf{realexp} \rightarrow \textbf{realexp}$.

To describe proper procedures similarly, it is necessary to introduce the type **comm** to describe phrases that are commands (or in Algol jargon, statements). Then a proper procedure accepting an integer variable would have type $\textbf{intvar} \rightarrow \textbf{comm}$.

The idea that commands are not a kind of expression is one of the things that distinguishes Algol-like languages from languages such as Scheme or ML, where commands are simply expressions that produce trivial values while performing side effects. This distinction is even sharper for Forsythe, where expressions cannot have side effects.

To extend the typing of procedures to permit more than one parameter, one might introduce a type constructor for products and regard, say, **procedure(intexp, intvar)** as (**intexp** × **intvar**) → **comm**. However, as we will see below, the product-like construction in Forsythe describes objects whose fields are selected by names rather than position. Thus multiple-parameter procedures are more easily obtained by Currying rather than by the use of products.

For example, **procedure(intexp, intvar)** becomes **intexp** → (**intvar** → **comm**) or, more simply, **intexp** → **intvar** → **comm**, since → is right associative. In other words, a proper procedure accepting an integer expression and an integer variable is really a procedure accepting an integer expression whose calls are procedures accepting an integer variable whose calls are commands. Thus the call $p(a_1, a_2)$ is written $(p(a_1))(a_2)$ or, more simply, $p(a_1)(a_2)$, since procedure application is left associative. (In fact, if the parameters are identifiers or constants, one can simply write $p\, a_1\, a_2$.)

In general, the type

$$\mathbf{procedure}(\theta_1, \ldots, \theta_n)$$

becomes

$$\theta_1 \to \cdots \to \theta_n \to \mathbf{comm}\,,$$

and, for each data type δ, the type

$$\delta\ \mathbf{procedure}(\theta_1, \ldots, \theta_n)$$

becomes

$$\theta_1 \to \cdots \to \theta_n \to \delta\mathbf{exp}\,.$$

Moreover, this generalization includes the special case where $n = 0$, so that parameterless proper procedures are simply commands and parameterless function procedures are simply expressions. (Note that this simplification is permissible for call by name, but would not be for call by value, where parameterless procedures are needed — as in LISP — to postpone evaluation.)

To determine the subtype relation for procedural types, suppose $\theta_1' \leq \theta_1$ and $\theta_2 \leq \theta_2'$. Then a procedure of type $\theta_1 \to \theta_2$ can accept a parameter of type θ_1' (since this parameter can be converted to type θ_1) and its call can have type θ_2' (since it can be converted from θ_2 to θ_2'), so that the procedure also has type $\theta_1' \to \theta_2'$. Thus

$$\text{If } \theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2'\,,$$

i.e. → is antimonotone in its first operand and monotone in its second operand.

For example, since **intexp** ≤ **realexp**, we have

$$\begin{tikzcd}
& \mathbf{intexp} \to \mathbf{realexp} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathbf{intexp} \to \mathbf{intexp} \arrow[dr, dash] & & \mathbf{realexp} \to \mathbf{realexp} \arrow[dl, dash] \\
& \mathbf{realexp} \to \mathbf{intexp} &
\end{tikzcd}$$

We have already seen that **comm**(and) must be introduced as a primitive phrase type. It is also useful to introduce a subtype of **comm** called **compl**(etion):

$$\begin{tikzcd}
\mathbf{comm} \arrow[d, dash] \\
\mathbf{compl}
\end{tikzcd}$$

Essentially, a completion is a special type of command, such as a **goto** command, that never returns control.

The advantage of distinguishing completions is that control structure can be made more evident. For example, in

> **procedure** *sqroot*(x, y, *error*); **intexp** x; **intvar** y; **compl** *error*;
>   **begin if** $x < 0$ **then** *error*; C **end** ,

specifying *error* to be a completion makes it evident that C will never be executed when $x < 0$.

As mentioned earlier, Forsythe has a type constructor for named products. The basic idea is that the phrase type

$$(\iota_1 \colon \theta_1, \ldots, \iota_n \colon \theta_n)$$

is possessed by objects with fields named by the distinct identifiers $\iota_1, \ldots, \iota_n$, in which the field named ι_k has type θ_k. Note that the meaning of this phrase type is independent of the order of the $\iota_k \colon \theta_k$ pairs. We use the term "object" rather than "record" since fields need not be variables. For example, one could have a field of type **intvar** → **comm** that could be called as a proper procedure, but not assigned to. (Roughly speaking, objects are more like class members in Simula 67 [11] than like records in Algol W [4].)

Clearly, the product constructor should be monotone:

$$\text{If } n \geq 0 \text{ and } \theta_1 \leq \theta_1' \text{ and } \ldots \text{ and } \theta_n \leq \theta_n' \text{ then}$$

$$(\iota_1 \colon \theta_1, \ldots, \iota_n \colon \theta_n) \leq (\iota_1 \colon \theta_1', \ldots, \iota_n \colon \theta_n') \,.$$

In fact, a richer subtype relationship is desirable, in which objects can be converted by "forgetting" fields, so that an object can be used in a context requiring a subset of its fields. This relationship (which is closely related to "multiple inheritance" in object-oriented programming [12]) is expressed by

$$\text{If } n \geq m \geq 0 \text{ and } \theta_1 \leq \theta_1' \text{ and } \ldots \text{ and } \theta_m \leq \theta_m' \text{ then}$$

$$(\iota_1 \colon \theta_1, \ldots, \iota_n \colon \theta_n) \leq (\iota_1 \colon \theta_1', \ldots, \iota_m \colon \theta_m') \,.$$

For example,

$$\begin{tikzcd}
\mathit{partno} \colon \mathbf{int} \arrow[dr, dash] & & \mathit{cost} \colon \mathbf{real} \arrow[dl, dash] \\
& (\mathit{partno} \colon \mathbf{int}, \mathit{cost} \colon \mathbf{real}) \arrow[d, dash] & \\
& (\mathit{partno} \colon \mathbf{intvar}, \mathit{cost} \colon \mathbf{realvar}) &
\end{tikzcd}$$

At this point, we have summarized the type structure of Forsythe (then called "Idealized Algol") as it appeared in about 1981 [5]. Since then, the language has been generalized, and considerably simplified, by the introduction of intersection types [13, 14, 15].

(At the outset, a caution must be sounded that this use of the word "intersection" can be misleading. If one thinks of types as standing for sets, than the intersection of two types need not stand for the intersection of the two corresponding sets. In earlier papers, we used the term "conjunctive type", but this was equally misleading in other contexts, and never became widely accepted.)

The basic idea is to introduce a type constructor &, with the interpretation that a phrase has type $\theta_1 \,\&\, \theta_2$ if and only if it has both type θ_1 and type θ_2. This interpretation leads to the subtype laws

$$\theta_1 \,\&\, \theta_2 \leq \theta_1$$

$$\theta_1 \,\&\, \theta_2 \leq \theta_2$$

$$\text{If } \theta \leq \theta_1 \text{ and } \theta \leq \theta_2 \text{ then } \theta \leq \theta_1 \,\&\, \theta_2 \,,$$

which assert that $\theta_1 \,\&\, \theta_2$ is a greatest lower bound of θ_1 and θ_2. (Note that the introduction of the intersection operation makes the subtype relation a preorder rather than a partial order, since one can have distinct types, such as $\theta_1 \,\&\, \theta_2$ and $\theta_2 \,\&\, \theta_1$, each of which is a subtype of the other. In this situation, we will say that the types are *equivalent.*)

We will see that intersection types provide the ability to define procedures with more than one type. For example

$$\mathbf{procedure}\ \mathit{poly}(x);\ x \times x + 2$$

can be given the type (**intexp** → **intexp**) & (**realexp** → **realexp**). At present, however, the main point is that intersection can be used to simplify the structure of types.

First, the various types of variables can be regarded as intersections of expressions and acceptors. For example, **intvar** is **intexp** & **intacc**, **realvar** is **realexp** & **realacc**, and (**int**, **real**) **var** is **realexp** & **intacc**.

Second, a product type with more than one field can be regarded as an intersection of product types with single fields. Thus, instead of

$$(\iota_1 \colon \theta_1, \ldots, \iota_n \colon \theta_n) \, ,$$

one writes

$$\iota_1 \colon \theta_1 \mathbin{\&} \cdots \mathbin{\&} \iota_n \colon \theta_n \, .$$

Note that the field-forgetting relationship becomes a consequence of $\theta_1 \mathbin{\&} \theta_2 \leq \theta_i$.

A final simplification concerns acceptors. The meaning of a δ acceptor a (for any data type δ) is completely determined by the meanings of the commands $a := e$ for all δ expressions e. Thus a has the same kind of meaning as a procedure of type δ**exp** → **comm**. As a consequence, we can regard δ**acc** as an abbreviation for δ**exp** → **comm**, and $a := e$ as an abbreviation for $a(e)$. (As discussed in Section 8, this treatment of assignment as procedure call is a controversial generalization of the usual concept of assignment.)

3. Types and the Subtype Relation

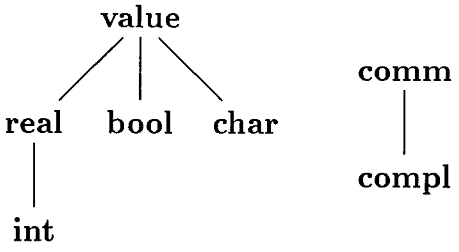
Having sketched its evolution, we can now define the type system of Forsythe precisely. The sets of data types, primitive (phrase) types, and (phrase) types can be defined by an abstract grammar:

$$\begin{array}{ll} \delta ::= \mathbf{int} \mid \mathbf{real} \mid \mathbf{bool} \mid \mathbf{char} & \text{(data types)} \\ \rho ::= \delta \mid \mathbf{value} \mid \mathbf{comm} \mid \mathbf{compl} & \text{(primitive types)} \\ \theta ::= \rho \mid \theta \to \theta \mid \iota \colon \theta \mid \mathbf{ns} \mid \theta \mathbin{\&} \theta & \text{(types)} \end{array}$$

where the metavariable ι ranges over identifiers.

Here there are three changes from the previous section. Expression types are now named by their underlying data types; for example, **intexp** is now just **int**. A new primitive type **value** stands for the union of all the data types; its utility will become apparent in Section 4. Finally, a new phrase type **ns** (for "nonsense") has been introduced; it is possessed by all (parsable) phrases of the language, and can be viewed as a unit for the operation &, i.e. as the intersection of the empty set of types.

The subtype relation $\leq_{\text{prim}}$ for primitive types is the partial order



For types, $\leq$ is the least preorder such that

$$\theta \leq \mathbf{ns}$$

$$\theta_1 \,\&\, \theta_2 \leq \theta_1$$

$$\theta_1 \,\&\, \theta_2 \leq \theta_2$$

$$\text{If } \theta \leq \theta_1 \text{ and } \theta \leq \theta_2 \text{ then } \theta \leq \theta_1 \,\&\, \theta_2$$

$$\text{If } \rho \leq_{\text{prim}} \rho' \text{ then } \rho \leq \rho'$$

$$\text{If } \theta \leq \theta' \text{ then } \iota{:}\,\theta \leq \iota{:}\,\theta'$$

$$\text{If } \theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2'$$

$$\iota{:}\,\theta_1 \,\&\, \iota{:}\,\theta_2 \leq \iota{:}\,(\theta_1 \,\&\, \theta_2)$$

$$(\theta \to \theta_1) \,\&\, (\theta \to \theta_2) \leq \theta \to (\theta_1 \,\&\, \theta_2)$$

$$\mathbf{ns} \leq \iota{:}\,\mathbf{ns}$$

$$\mathbf{ns} \leq \theta \to \mathbf{ns} \ .$$

We write $\theta \simeq \theta'$, and say that θ and θ' are equivalent, when $\theta \leq \theta'$ and $\theta' \leq \theta$. The first four relationships establish that **ns** is a greatest type and that $\theta_1 \,\&\, \theta_2$ is a greatest lower bound of θ_1 and θ_2. Note that we say "a" rather than "the"; neither greatest types nor greatest lower bounds are unique, since we have a preorder rather than a partial order, However, any greatest type must be equivalent to **ns**, and any greatest lower bound of θ_1 and θ_2 must be equivalent to $\theta_1 \,\&\, \theta_2$.

The fact that **ns** is a greatest type and & is a greatest lower bound operator has the following consequences:

$$\theta_1 \,\&\, (\theta_2 \,\&\, \theta_3) \simeq (\theta_1 \,\&\, \theta_2) \,\&\, \theta_3$$

$$\theta \,\&\, \mathbf{ns} \simeq \theta$$

$$\mathbf{ns} \,\&\theta \simeq \theta$$

$$\theta_1 \,\&\, \theta_2 \simeq \theta_2 \,\&\, \theta_1$$

$$\theta \,\&\, \theta \simeq \theta$$

$$\text{If } \theta_1 \leq \theta_1' \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \,\&\, \theta_2 \leq \theta_1' \,\&\, \theta_2'$$

$$\theta \leq \theta_1 \,\&\, \theta_2 \text{ iff } \theta \leq \theta_1 \text{ and } \theta \leq \theta_2 \ .$$

The next three relationships in the definition of $\leq$ assert that primitive types are related by $\leq_{\text{prim}}$, that the object-type constructor is monotone, and that $\to$ is antimonotone in its first operand and monotone in its second operand. The last four relationships have the following consequences:

$$\iota{:}\,(\theta_1 \,\&\, \theta_2) \simeq \iota{:}\,\theta_1 \,\&\, \iota{:}\,\theta_2$$

$$\theta \to (\theta_1 \,\&\, \theta_2) \simeq (\theta \to \theta_1) \,\&\, (\theta \to \theta_2)$$

$$\iota\colon \mathbf{ns} \simeq \mathbf{ns}$$

$$\theta \to \mathbf{ns} \simeq \mathbf{ns}\ .$$

The first two of these equivalences show that intersection distributes with object constructors (modulo $\simeq$) and with the right side (but not the left) of $\to$. The last two equivalences are analogous laws for the intersection of zero types.

It can be shown that every pair of types has a least upper bound (which is unique modulo $\simeq$). In particular, the following equivalences suffice to compute a least upper bound, $\theta_1 \sqcup \theta_2$, of any types θ_1 and θ_2:

$$\theta_1 \sqcup \theta_2 \simeq \theta_2 \sqcup \theta_1$$

$$\theta \sqcup \mathbf{ns} \simeq \mathbf{ns}$$

$$\theta_1 \sqcup (\theta_2 \mathbin{\&} \theta_3) \simeq (\theta_1 \sqcup \theta_2) \mathbin{\&} (\theta_1 \sqcup \theta_3)$$

$$\rho \sqcup \iota\colon \theta \simeq \mathbf{ns}$$

$$\rho \sqcup (\theta_1 \to \theta_2) \simeq \mathbf{ns}$$

$$\iota\colon \theta_1 \sqcup (\theta_2 \to \theta_3) \simeq \mathbf{ns}$$

$$\rho_1 \sqcup \rho_2 \simeq \rho_1 \sqcup_{\text{prim}} \rho_2 \text{ when } \rho_1 \sqcup_{\text{prim}} \rho_2 \text{ exists}$$

$$\rho_1 \sqcup \rho_2 \simeq \mathbf{ns} \text{ when } \rho_1 \sqcup_{\text{prim}} \rho_2 \text{ does not exist}$$

$$\iota\colon \theta_1 \sqcup \iota\colon \theta_2 \simeq \iota\colon (\theta_1 \sqcup \theta_2)$$

$$\iota_1\colon \theta_1 \sqcup \iota_2\colon \theta_2 \simeq \mathbf{ns} \text{ when } \iota_1 \neq \iota_2$$

$$(\theta_1 \to \theta_1') \sqcup (\theta_2 \to \theta_2') \simeq (\theta_1 \mathbin{\&} \theta_2) \to (\theta_1' \sqcup \theta_2')\ .$$

The types **int**, **real**, **bool**, **char**, **value**, **comm**, **compl**, and **ns** are actually predefined *type identifiers* (whose meaning can be redefined by the **lettype** definition to be discussed later, but which take on standard meanings outside of such redefinitions). Additional predefined type identifiers are provided to abbreviate various commonly occurring nonprimitive types. As discussed in the previous section, when δ is any of the character sequences **int**, **real**, **bool**, or **char** that denote data types,

$$\delta\mathbf{acc} \stackrel{\text{def}}{=} \delta \to \mathbf{comm}$$

(e.g. $\mathbf{intacc} \stackrel{\text{def}}{=} \mathbf{int} \to \mathbf{comm}$), and

$$\delta\mathbf{var} \stackrel{\text{def}}{=} \delta \mathbin{\&} \delta\mathbf{acc}\ .$$

There are also abbreviations for commonly occurring types of sequences. In general, a sequence s of element type θ and length n is an entity of type $(\mathbf{int} \to \theta) \mathbin{\&} \mathit{len}\colon \mathbf{int}$ such that the value of $s.\mathit{len}$ is n and the application $s\,i$ is well-defined for all integers i such that $0 \leq i < n$. (Of course, the proviso on definedness is not implied by the type of the sequence.)

The following type identifiers are predefined to abbreviate specific types of sequences:

$$\delta\mathbf{seq} \overset{\text{def}}{=} (\mathbf{int} \to \delta) \mathbin{\&} \mathit{len}\colon\mathbf{int}$$

$$\delta\mathbf{accseq} \overset{\text{def}}{=} (\mathbf{int} \to \delta\mathbf{acc}) \mathbin{\&} \mathit{len}\colon\mathbf{int}$$

$$\delta\mathbf{varseq} \overset{\text{def}}{=} (\mathbf{int} \to \delta\mathbf{var}) \mathbin{\&} \mathit{len}\colon\mathbf{int}$$

$$\mathbf{commseq} \overset{\text{def}}{=} (\mathbf{int} \to \mathbf{comm}) \mathbin{\&} \mathit{len}\colon\mathbf{int}$$

$$\mathbf{complseq} \overset{\text{def}}{=} (\mathbf{int} \to \mathbf{compl}) \mathbin{\&} \mathit{len}\colon\mathbf{int}\ .$$

For instance, a δ**varseq**, in Algol terminology, is a one-dimensional δ array with a lower bound of zero and an upper bound one less than its length. A δ**seq** is a similar entity whose elements can be evaluated but not assigned to, and a δ**accseq** is a similar entity whose elements can be assigned to but not evaluated. For example, **charseq** is the type of string constants.

4. The Semantics of Types

To describe the meaning of types, we will employ some basic concepts from category theory. The main reason for doing so is that, by formulating succinct definitions in terms of a mathematical theory of great generality, we gain an assurance that our language will be uniform and general.

A second reason is that the abstract concept of a category establishes a bridge between intuitive and rigorous semantics. Intuitively, we think of a type as standing for a set, and an implicit conversion as a function from one such set to another. But since our language permits nonterminating programs, types must denote domains (i.e. complete partial orders with a least element) and implicit conversions must be continuous functions. Moreover, a further level of complication arises when one develops a semantics that embodies the block structure of Algol-like languages; then types denote functors and implicit conversions are natural transformations between such functors [5, 16, 17].

However, the choice between these three different views is simply a choice between three different "semantic" categories:

- SET — in which the objects are sets, and the set of morphisms $S \to S'$ is the set of functions from S to S'.
- DOM — in which the objects are domains, and $D \to D'$ is the set of continuous functions from D to D'.
- PDOM^{Σ} — in which the objects are functors from a category Σ of "store shapes" to the category PDOM of predomains (complete partial orders, possibly without least elements) and continuous functions, and $F \to F'$ is the set of natural transformations from F to F'.

Therefore, if we formulate the semantics of types in terms of an arbitrary category, assuming only properties that are possessed by all three of the above categories (i.e. being Cartesian closed and possessing certain limits), then we can think about the semantics in the intuitive setting of sets and functions, yet be confident that our semantics makes sense in a more rigorous setting.

Thus we will define types in terms of an unspecified semantic category, while giving explanatory remarks and examples in terms of the particular category SET (or occasionally DOM).

For each type θ, we write $\llbracket\theta\rrbracket$ for the object (e.g. set) denoted by θ. Whenever $\theta \leq \theta'$, we write $\llbracket\theta \leq \theta'\rrbracket$ for the implicit conversion morphism (e.g. function) from $\llbracket\theta\rrbracket$ to $\llbracket\theta'\rrbracket$. Two requirements are imposed on these implicit conversion morphisms:

- For all types θ, the conversion from $\llbracket\theta\rrbracket$ to $\llbracket\theta\rrbracket$ must be an identity:
$$\llbracket\theta \leq \theta\rrbracket = I_{\llbracket\theta\rrbracket} \,.$$
- Whenever $\theta \leq \theta'$ and $\theta' \leq \theta''$, the composition of $\llbracket\theta \leq \theta'\rrbracket$ with $\llbracket\theta' \leq \theta''\rrbracket$ must equal $\llbracket\theta \leq \theta''\rrbracket$, i.e. the diagram
$$\begin{tikzcd}
\llbracket\theta\rrbracket \arrow[r, "\llbracket\theta \leq \theta'\rrbracket"] \arrow[dr, "\llbracket\theta \leq \theta''\rrbracket"'] & \llbracket\theta'\rrbracket \arrow[d, "\llbracket\theta' \leq \theta''\rrbracket"] \\
 & \llbracket\theta''\rrbracket
\end{tikzcd}$$
must commute.

These requirements coincide with a basic concept of category theory: $\llbracket-\rrbracket$ must be a functor from the preordered set of types (viewed as a category) to the semantic category.

The above requirements determine the semantics of equivalence. When $\theta \simeq \theta'$, the diagrams
$$\begin{tikzcd}
\llbracket\theta\rrbracket \arrow[r, "\llbracket\theta \leq \theta'\rrbracket"] \arrow[dr, "\llbracket\theta \leq \theta\rrbracket = I_{\llbracket\theta\rrbracket}"'] & \llbracket\theta'\rrbracket \arrow[d, "\llbracket\theta' \leq \theta\rrbracket"] \\
 & \llbracket\theta\rrbracket
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
\llbracket\theta'\rrbracket \arrow[r, "\llbracket\theta' \leq \theta\rrbracket"] \arrow[dr, "\llbracket\theta' \leq \theta'\rrbracket = I_{\llbracket\theta'\rrbracket}"'] & \llbracket\theta\rrbracket \arrow[d, "\llbracket\theta \leq \theta'\rrbracket"] \\
 & \llbracket\theta'\rrbracket
\end{tikzcd}$$
both commute, so that $\llbracket\theta\rrbracket$ and $\llbracket\theta'\rrbracket$ are *isomorphic*, which we denote by $\llbracket\theta\rrbracket \approx \llbracket\theta'\rrbracket$. (Note, however, that nonequivalent types may also denote isomorphic objects.)

Next, we define (up to isomorphism) the meaning of each type constructor:

- Procedures — To define $\rightarrow$, we require the semantic category to be Cartesian closed, and define $\llbracket \theta \rightarrow \theta' \rrbracket$ to be $\llbracket \theta \rrbracket \Rightarrow \llbracket \theta' \rrbracket$, where $\Rightarrow$ denotes the exponentiation operation in the semantic category. In SET (DOM), $\llbracket \theta \rrbracket \Rightarrow \llbracket \theta' \rrbracket$ is the set (domain) of all (continuous) functions from $\llbracket \theta \rrbracket$ to $\llbracket \theta' \rrbracket$.

- Object Constructors — We define $\llbracket \iota \colon \theta \rrbracket$ to be an object that is isomorphic to $\llbracket \theta \rrbracket$.

- Nonsense — We define $\llbracket \mathbf{ns} \rrbracket$ to be a terminal object $\top$, i.e. an object such that, for any object s, there is exactly one morphism from s to $\top$. In SET or DOM a terminal object is a set containing one element. (Thus even nonsense phrases have a meaning, but they all have the same meaning.)

- Intersection — Because of its novelty, we describe the meaning of intersection in more detail than the other type constructors. Basically, the meaning of $\theta_1 \,\&\, \theta_2$ is determined by the meanings of θ_1, θ_2, and their least upper bound $\theta_1 \sqcup \theta_2$. From $\theta_1 \,\&\, \theta_2$, we can convert to θ_1 and from there to $\theta_1 \sqcup \theta_2$, or we can convert to θ_2 and from there to $\theta_1 \sqcup \theta_2$; clearly the two compositions of conversions should be equal. Moreover, whenever $\theta \leq \theta_1 \,\&\, \theta_2$, the composite composition from θ to $\theta_1 \,\&\, \theta_2$ to θ_1 should equal the direct conversion from θ to θ_1, and similarly for θ_2. In other words, in the diagram

$$\begin{tikzcd}
 & \llbracket \theta_1 \sqcup \theta_2 \rrbracket & \\
\llbracket \theta_1 \rrbracket \arrow[ur, "\llbracket \theta_1 \leq \theta_1 \sqcup \theta_2 \rrbracket"] & & \llbracket \theta_2 \rrbracket \arrow[ul, "\llbracket \theta_2 \leq \theta_1 \sqcup \theta_2 \rrbracket"'] \\
 & \llbracket \theta_1 \,\&\, \theta_2 \rrbracket \arrow[ul, "\llbracket \theta_1 \,\&\, \theta_2 \leq \theta_1 \rrbracket"'] \arrow[ur, "\llbracket \theta_1 \,\&\, \theta_2 \leq \theta_2 \rrbracket"] & \\
 & \llbracket \theta \rrbracket \arrow[u] \arrow[uul, "\llbracket \theta \leq \theta_1 \rrbracket"] \arrow[uur, "\llbracket \theta \leq \theta_2 \rrbracket"'] &
\end{tikzcd}$$

the inner diamond must commute and, for all θ such that $\theta \leq \theta_1 \,\&\, \theta_2$, the two triangles must commute.

However, these requirements are not sufficient to determine $\llbracket \theta_1 \,\&\, \theta_2 \rrbracket$. To strengthen them, we replace $\llbracket \theta \rrbracket$ by an arbitrary object s and $\llbracket \theta \leq \theta_1 \rrbracket$ and $\llbracket \theta \leq \theta_2 \rrbracket$ by any functions

f_1 and f_2 that make the outer diamond commute, and we require the "mediating morphism" from s to $\llbracket\theta_1 \& \theta_2\rrbracket$ to be unique. Specifically, we define $\llbracket\theta_1 \& \theta_2\rrbracket$ by requiring that, in the diagram

$$\begin{tikzcd}
 & \llbracket\theta_1 \sqcup \theta_2\rrbracket & \\
\llbracket\theta_1\rrbracket \arrow[ur, "\llbracket\theta_1 \leq \theta_1 \sqcup \theta_2\rrbracket"] & & \llbracket\theta_2\rrbracket \arrow[ul, "\llbracket\theta_2 \leq \theta_1 \sqcup \theta_2\rrbracket"'] \\
 & \llbracket\theta_1 \& \theta_2\rrbracket \arrow[ul, "\llbracket\theta_1 \& \theta_2 \leq \theta_1\rrbracket"] \arrow[ur, "\llbracket\theta_1 \& \theta_2 \leq \theta_2\rrbracket"'] & \\
 & s \arrow[uul, "f_1"] \arrow[uur, "f_2"'] \arrow[u, dashed] &
\end{tikzcd}$$

the inner diamond must commute and, for all objects s and morphisms f_1 and f_2 that make the outer diamond commute, there must be a unique morphism from s to $\llbracket\theta_1 \& \theta_2\rrbracket$ that makes the two triangles commute.

Clearly, this strengthening is something of a leap of faith. Thus it is reassuring that our definition coincides with a standard concept of category theory: we have defined $\llbracket\theta_1 \& \theta_2\rrbracket$ to be the *pullback* of $\llbracket\theta_1\rrbracket$, $\llbracket\theta_2\rrbracket$, and $\llbracket\theta_1 \sqcup \theta_2\rrbracket$ (which is unique up to isomorphism).

For sets or domains, the pullback is

$$\llbracket\theta_1 \& \theta_2\rrbracket \approx \Big\{\, \langle x_1, x_2\rangle \;\Big|\; x_1 \in \llbracket\theta_1\rrbracket \text{ and } x_2 \in \llbracket\theta_2\rrbracket \text{ and } \llbracket\theta_1 \leq \theta_1 \sqcup \theta_2\rrbracket x_1 = \llbracket\theta_2 \leq \theta_1 \sqcup \theta_2\rrbracket x_2 \,\Big\}.$$

(For domains, one must require all implicit conversion functions to be strict.) In other words, a meaning of type $\theta_1 \& \theta_2$ is a meaning of type θ_1 paired with a meaning of type θ_2, subject to the constraint that these meanings must convert to the same meaning of type $\theta_1 \sqcup \theta_2$.

The following are special cases of the definition of intersection. Although we describe these cases in terms of SET and DOM, basically similar results hold for any semantic category that is Cartesian closed and possesses the pullbacks necessary to define intersection.

- If $\theta_1 \sqcup \theta_2 \simeq \mathbf{ns}$ then the constraint

  $$[\![\theta_1 \leq \theta_1 \sqcup \theta_2]\!]x_1 = [\![\theta_2 \leq \theta_1 \sqcup \theta_2]\!]x_2$$

  always holds, since both sides of the equation belong to the one-element set $[\![\mathbf{ns}]\!]$. Thus

  $$[\![\theta_1 \,\&\, \theta_2]\!] \approx [\![\theta_1]\!] \times [\![\theta_2]\!] \,.$$

  For example,

  $$[\![\mathbf{intvar}]\!] = [\![\mathbf{int}\,\&(\mathbf{int} \rightarrow \mathbf{comm})]\!] \approx [\![\mathbf{int}]\!] \times [\![\mathbf{int} \rightarrow \mathbf{comm}]\!]$$

  $$[\![\iota\colon \theta_1 \,\&\, (\theta_2 \rightarrow \theta_3)]\!] \approx [\![\iota\colon \theta_1]\!] \times [\![\theta_2 \rightarrow \theta_3]\!] \approx [\![\theta_1]\!] \times [\![\theta_2 \rightarrow \theta_3]\!]$$

  and, when $\iota_1 \neq \iota_2$,

  $$[\![\iota_1\colon \theta_1 \,\&\, \iota_2\colon \theta_2]\!] \approx [\![\iota_1\colon \theta_1]\!] \times [\![\iota_2\colon \theta_2]\!] \approx [\![\theta_1]\!] \times [\![\theta_2]\!] \,.$$

- If $\theta_1 \leq \theta_2$, so that $\theta_1 \sqcup \theta_2 = \theta_2$, then

  $$[\![\theta_1 \,\&\, \theta_2]\!] \approx \left\{ \langle x_1, x_2 \rangle \;\middle|\; x_1 \in [\![\theta_1]\!] \text{ and } x_2 \in [\![\theta_2]\!] \text{ and } [\![\theta_1 \leq \theta_2]\!]x_1 = x_2 \right\} \approx [\![\theta_1]\!] \,.$$

  For example,

  $$[\![\mathbf{int}\,\&\,\mathbf{real}]\!] \approx [\![\mathbf{int}]\!]$$

  $$[\![\mathbf{compl}\,\&\,\mathbf{comm}]\!] \approx [\![\mathbf{compl}]\!] \,.$$

- If $[\![\theta_1]\!]$ and $[\![\theta_2]\!]$ are subsets of $[\![\theta_1 \sqcup \theta_2]\!]$, and $[\![\theta_1 \leq \theta_1 \sqcup \theta_2]\!]$ and $[\![\theta_2 \leq \theta_1 \sqcup \theta_2]\!]$ are identity injections, then

  $$[\![\theta_1 \,\&\, \theta_2]\!] \approx \left\{ \langle x_1, x_2 \rangle \;\middle|\; x_1 \in [\![\theta_1]\!] \text{ and } x_2 \in [\![\theta_2]\!] \text{ and } x_1 = x_2 \right\} \approx [\![\theta_1]\!] \cap [\![\theta_2]\!] \,.$$

  In this special case, the intersection of types does correspond to the intersection of sets.

  An example of this case arises when θ_1 and θ_2 are data types with no implicit conversion between them, such as **int** and **char**. In this case, their least upper bound is **value**, which stands for the union of the data types, so that the implicit conversions into **value** are identity injections. Thus,

  $$[\![\mathbf{int}\,\&\,\mathbf{char}]\!] \approx [\![\mathbf{int}]\!] \cap [\![\mathbf{char}]\!] \,.$$

  This is the purpose of introducing the type **value**. Had we not done so, we would have $\mathbf{int} \sqcup \mathbf{char} = \mathbf{ns}$, which would give $[\![\mathbf{int}\,\&\,\mathbf{char}]\!] \approx [\![\mathbf{int}]\!] \times [\![\mathbf{char}]\!]$.

  In Forsythe, the sets denoted by the data types **real**, **bool**, and **char** are disjoint (and the set denoted by **int** is a subset of that denoted by **real**), so that intersections such as **int** & **char** denote the empty set. However, this is a detail of the language design, while the preceding argument is more general.

The above arguments are based on the intuition that data types denote sets of values. In fact, however, data types such as **int** and **char**, when they are used as phrase types, denote sets of meanings appropriate to kinds of expressions. Specifically, in a domain-theoretic model,

$$\llbracket \mathbf{int} \rrbracket = S \to Z_\perp \qquad \llbracket \mathbf{char} \rrbracket = S \to C_\perp \qquad \llbracket \mathbf{value} \rrbracket = S \to V_\perp \,,$$

where S is the set of states; $Z_\perp$, $C_\perp$, and $V_\perp$ are set of integers, characters, and values, made into flat domains by adding a least element; and Z and C are subsets of V. Even in this richer setting, however, it is still true that the conversion functions from $\llbracket \mathbf{int} \rrbracket$ and $\llbracket \mathbf{char} \rrbracket$ into $\llbracket \mathbf{value} \rrbracket$ are identity injections.

- Finally, we consider the intersection of procedural types. First, we must define the implicit conversions between such types. If $\theta_1' \leq \theta_1$ and $\theta_2 \leq \theta_2'$ then the conversion of $f \in \llbracket \theta_1 \to \theta_2 \rrbracket$ to $\llbracket \theta_1' \to \theta_2' \rrbracket$ is obtained by composing f with appropriate conversions of its arguments and results:

$$\begin{tikzcd}
\llbracket \theta_1' \rrbracket \arrow[r, "\llbracket \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2' \rrbracket f"] \arrow[d, "\llbracket \theta_1' \leq \theta_1 \rrbracket"'] & \llbracket \theta_2' \rrbracket \\
\llbracket \theta_1 \rrbracket \arrow[r, "f"] & \llbracket \theta_2 \rrbracket \arrow[u, "\llbracket \theta_2 \leq \theta_2' \rrbracket"']
\end{tikzcd}$$

or as an equation,

$$\llbracket \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2' \rrbracket f = \llbracket \theta_1' \leq \theta_1 \rrbracket \,;\, f \,;\, \llbracket \theta_2 \leq \theta_2' \rrbracket \,,$$

where ; denotes composition in diagrammatic order.

From the definition of intersection, by substituting the equation for the least upper bound of two procedural types, and using the above equation for the implicit conversion of procedural types, we obtain

$$\begin{aligned}
&\llbracket (\theta_1 \to \theta_1') \,\&\, (\theta_2 \to \theta_2') \rrbracket \approx \\
&\Big\{ \langle f_1, f_2 \rangle \;\Big|\; f_1 \in \llbracket \theta_1 \to \theta_1' \rrbracket \text{ and } f_2 \in \llbracket \theta_2 \to \theta_2' \rrbracket \\
&\qquad \text{and } \llbracket \theta_1 \to \theta_1' \leq (\theta_1 \,\&\, \theta_2) \to (\theta_1' \sqcup \theta_2') \rrbracket f_1 = \llbracket \theta_2 \to \theta_2' \leq (\theta_1 \,\&\, \theta_2) \to (\theta_1' \sqcup \theta_2') \rrbracket f_2 \Big\} \\
&= \Big\{ \langle f_1, f_2 \rangle \;\Big|\; f_1 \in \llbracket \theta_1 \to \theta_1' \rrbracket \text{ and } f_2 \in \llbracket \theta_2 \to \theta_2' \rrbracket \\
&\qquad\quad \text{and } \llbracket \theta_1 \,\&\, \theta_2 \leq \theta_1 \rrbracket \,;\, f_1 \,;\, \llbracket \theta_1' \leq \theta_1' \sqcup \theta_2' \rrbracket = \llbracket \theta_1 \,\&\, \theta_2 \leq \theta_2 \rrbracket \,;\, f_2 \,;\, \llbracket \theta_2' \leq \theta_1' \sqcup \theta_2' \rrbracket \Big\} .
\end{aligned}$$

Here the constraint on f_1 and f_2 is the commutativity of a hexagon:

$$\begin{tikzcd}
 & \llbracket \theta_1 \rrbracket \arrow[r, "f_1"] & \llbracket \theta_1' \rrbracket \arrow[dr, "\llbracket \theta_1' \leq \theta_1' \sqcup \theta_2' \rrbracket"] & \\
\llbracket \theta_1 \,\&\, \theta_2 \rrbracket \arrow[ur, "\llbracket \theta_1 \,\&\, \theta_2 \leq \theta_1 \rrbracket"] \arrow[dr, "\llbracket \theta_1 \,\&\, \theta_2 \leq \theta_2 \rrbracket"'] & & & \llbracket \theta_1' \sqcup \theta_2' \rrbracket \\
 & \llbracket \theta_2 \rrbracket \arrow[r, "f_2"'] & \llbracket \theta_2' \rrbracket \arrow[ur, "\llbracket \theta_2' \leq \theta_1' \sqcup \theta_2' \rrbracket"'] &
\end{tikzcd}$$

This constraint implies that the "versions" of a procedure whose type is an intersection must respect implicit conversions — which is what distinguishes such a procedure from the usual notion of a "generic" procedure. For example, since **int** & **real** = **int** and **int** ⊔ **real** = **real** (taking = rather than ≃ here simplifies the argument),

$$\begin{aligned}&[\![(\mathbf{int} \to \mathbf{int})\ \&\ (\mathbf{real} \to \mathbf{real})]\!] \approx \\ &\Big\{ \langle f_1, f_2 \rangle \ \Big|\ f_1 \in [\![\mathbf{int} \to \mathbf{int}]\!] \text{ and } f_2 \in [\![\mathbf{real} \to \mathbf{real}]\!] \\ &\qquad\qquad \text{and } f_1 \,;\, [\![\mathbf{int} \le \mathbf{real}]\!] = [\![\mathbf{int} \le \mathbf{real}]\!] \,;\, f_2 \Big\}.\end{aligned}$$

Here the hexagon collapses into a rectangle, so that f_1 and f_2 must satisfy

$$\begin{tikzcd}
{[\![\mathbf{int}]\!]} \arrow[r, "f_1"] \arrow[d, "{[\![\mathbf{int} \le \mathbf{real}]\!]}"'] & {[\![\mathbf{int}]\!]} \arrow[d, "{[\![\mathbf{int} \le \mathbf{real}]\!]}"] \\
{[\![\mathbf{real}]\!]} \arrow[r, "f_2"] & {[\![\mathbf{real}]\!]}
\end{tikzcd}$$

On the other hand,

$$[\![(\mathbf{int} \to \mathbf{int})\ \&\ (\mathbf{char} \to \mathbf{char})]\!] \approx [\![\mathbf{int} \to \mathbf{int}]\!] \times [\![\mathbf{char} \to \mathbf{char}]\!]\,,$$

since in this case the hexagonal constraint on f_1 and f_2 is vacuously true because $[\![\mathbf{int}\ \&\ \mathbf{char}]\!]$ is the empty set.

5. Phrases and their Typings

We now introduce the phrases of Forsythe and give rules for determining their types. Specifically, we will give inference rules for formulas called typings.

A *type assignment* is a function from a finite set of identifiers to types. If π is a type assignment, then $[\,\pi \mid \iota{:}\,\theta\,]$ denotes the type assignment whose domain is $\operatorname{dom}\pi \cup \{\iota\}$, such that $[\,\pi \mid \iota{:}\,\theta\,]\iota = \theta$ and $[\,\pi \mid \iota{:}\,\theta\,]\iota' = \pi\iota'$ when $\iota' \neq \iota$. We write $[\,\pi \mid \iota_1{:}\,\theta_1 \mid \cdots \mid \iota_n{:}\,\theta_n\,]$ to abbreviate $[\cdots[\,\pi \mid \iota_1{:}\,\theta_1\,]\cdots \mid \iota_n{:}\,\theta_n\,]$.

If π is a type assignment, p is a phrase, and θ is a type, then the formula $\pi \vdash p : \theta$, called a *typing*, asserts that the phrase p has the type θ when its free identifiers are assigned types by π.

An *inference rule* consists of zero or more typings called *premisses* followed (after a horizontal line) by one or more typings called *conclusions*. The rule may contain metavariables denoting type assignments, phrases, identifiers, or types; an instance of the rule is obtained by replacing these metavariables by particular type assignments, phrases, identifiers, or types. (Some rules will have restrictions on the permissible replacements.) The meaning of a rule is that, for any instance, if all the premisses are valid typings then all of the conclusions are valid typings.

First, we have rules describing the behavior of subtypes, the nonsense type, and intersections of types:

- Subtypes (often called subsumption)

$$\frac{\pi \vdash p : \theta}{\pi \vdash p : \theta'} \quad \text{when } \theta \leq \theta'$$

- Nonsense

$$\frac{}{\pi \vdash p : \mathbf{ns}}$$

- Intersection

$$\frac{\begin{array}{l}\pi \vdash p : \theta_1 \\ \pi \vdash p : \theta_2\end{array}}{\pi \vdash p : \theta_1 \mathbin{\&} \theta_2}$$

Then there are rules for typing identifiers, applications (procedure calls), and conditional phrases:

- Identifiers

$$\frac{}{\pi \vdash \iota : \pi(\iota)} \quad \text{when } \iota \in \operatorname{dom} \pi$$

- Applications

$$\frac{\begin{array}{l}\pi \vdash p_1 : \theta \rightarrow \theta' \\ \pi \vdash p_2 : \theta\end{array}}{\pi \vdash p_1\, p_2 : \theta'}$$

- Conditionals

$$\frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{bool} \\ \pi \vdash p_2 : \theta \\ \pi \vdash p_3 : \theta\end{array}}{\pi \vdash \mathbf{if}\, p_1\, \mathbf{then}\, p_2\, \mathbf{else}\, p_3 : \theta}$$

Notice that the conditional construction is applicable to arbitrary types.

Next we consider abstractions (sometimes called lambda expressions), which are used to denote procedures. Here there are two cases, depending upon whether the type of the argument to the procedure is indicated explicitly. In the explicit case we have:

- Abstractions (with explicit typing)

$$\frac{[\,\pi \mid \iota{:}\,\theta_i\,] \vdash p : \theta'}{\pi \vdash (\lambda\iota{:}\,\theta_1 \mid \cdots \mid \theta_n.\, p) : \theta_i \rightarrow \theta'}$$

In this rule, notice that θ_i must be one of a list of types appearing explicitly in the abstraction (separated by the alternative operator $\mid$). For example, under any type assignment, the abstraction

$$\lambda x\colon \mathbf{int}.\ x \times x + 2$$

has type $\mathbf{int} \to \mathbf{int}$, while the abstraction

$$\lambda x\colon \mathbf{int} \mid \mathbf{real}.\ x \times x + 2$$

has both type $\mathbf{int} \to \mathbf{int}$ and type $\mathbf{real} \to \mathbf{real}$, so that, by the rule for intersection, it also has type $(\mathbf{int} \to \mathbf{int})\ \&\ (\mathbf{real} \to \mathbf{real})$. (Note the role of the colon, which is always used in Forsythe to specify the types of identifiers or phrases.)

In contrast, in the case where the argument is implicitly typed, we have:

- Abstractions (with implicit typing)

$$\frac{[\pi \mid \iota\colon \theta] \vdash p : \theta'}{\pi \vdash (\lambda\iota.\ p) : \theta \to \theta'}$$

Here the abstraction provides no explicit constraints on the type θ. For example, for the abstraction $\lambda x.\ x \times x + 2$, one can use this rule to infer either of the types $\mathbf{int} \to \mathbf{int}$ or $\mathbf{real} \to \mathbf{real}$. More vividly, for the abstraction $\lambda x.\ x$, one can infer any type of the form $\theta \to \theta$.

At this point, one might ask why one would ever use explicit typing. Sensible answers are to make the program more readable, or to insure that a procedure has the typing one expects, rather than just some typing that makes the overall program type-correct. But a more stringent answer is that it has been proven that there is no algorithm that can typecheck an arbitrary implicitly typed program in the intersection type discipline [13, 14, 15]. Thus the Forsythe implementation requires some explicit type information to be provided. The exact nature of this requirement is described in Appendix C.

Next there are constructions for denoting objects and selecting their fields:

- Object Construction

$$\frac{\pi \vdash p : \theta}{\pi \vdash (\iota \equiv p) : (\iota\colon \theta)}$$

- Field Selection

$$\frac{\pi \vdash p : (\iota\colon \theta)}{\pi \vdash p.\iota : \theta}$$

The first of these forms denotes objects with only a single field; objects with several fields can be denoted by the merge construction, which will be described later. Note the role of the connective $\equiv$, which is always used to connect identifiers with their meanings.

Then comes a long list of rules describing various types of constants and expressions:

- Constants

$$\frac{}{\pi \vdash \langle\text{nat const}\rangle : \mathbf{int}}$$

$$\frac{}{\pi \vdash \langle\text{real const}\rangle : \mathbf{real}}$$

$$\frac{}{\pi \vdash \langle\text{char const}\rangle : \mathbf{char}}$$

$$\frac{}{\pi \vdash \langle\text{string const}\rangle : \mathbf{charseq}}$$

- Arithmetic Expressions

$$\frac{\pi \vdash p : \mathbf{int}}{\begin{array}{l}\pi \vdash +p : \mathbf{int}\\ \pi \vdash -p : \mathbf{int}\end{array}} \qquad \frac{\pi \vdash p : \mathbf{real}}{\begin{array}{l}\pi \vdash +p : \mathbf{real}\\ \pi \vdash -p : \mathbf{real}\end{array}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{int}\\ \pi \vdash p_2 : \mathbf{int}\end{array}}{\begin{array}{l}\pi \vdash p_1 + p_2 : \mathbf{int}\\ \pi \vdash p_1 - p_2 : \mathbf{int}\\ \pi \vdash p_1 \times p_2 : \mathbf{int}\\ \pi \vdash p_1 \div p_2 : \mathbf{int}\\ \pi \vdash p_1 \,\mathbf{rem}\, p_2 : \mathbf{int}\\ \pi \vdash p_1 ** p_2 : \mathbf{int}\end{array}} \qquad \frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{real}\\ \pi \vdash p_2 : \mathbf{real}\end{array}}{\begin{array}{l}\pi \vdash p_1 + p_2 : \mathbf{real}\\ \pi \vdash p_1 - p_2 : \mathbf{real}\\ \pi \vdash p_1 \times p_2 : \mathbf{real}\\ \pi \vdash p_1 / p_2 : \mathbf{real}\end{array}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{real}\\ \pi \vdash p_2 : \mathbf{int}\end{array}}{\pi \vdash p_1 \uparrow p_2 : \mathbf{real}}$$

- Relations

$$\frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{real}\\ \pi \vdash p_2 : \mathbf{real}\end{array}}{\begin{array}{l}\pi \vdash p_1 = p_2 : \mathbf{bool}\\ \pi \vdash p_1 \neq p_2 : \mathbf{bool}\\ \pi \vdash p_1 < p_2 : \mathbf{bool}\\ \pi \vdash p_1 \leq p_2 : \mathbf{bool}\\ \pi \vdash p_1 > p_2 : \mathbf{bool}\\ \pi \vdash p_1 \geq p_2 : \mathbf{bool}\end{array}} \qquad \frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{char}\\ \pi \vdash p_2 : \mathbf{char}\end{array}}{\begin{array}{l}\pi \vdash p_1 = p_2 : \mathbf{bool}\\ \pi \vdash p_1 \neq p_2 : \mathbf{bool}\\ \pi \vdash p_1 < p_2 : \mathbf{bool}\\ \pi \vdash p_1 \leq p_2 : \mathbf{bool}\\ \pi \vdash p_1 > p_2 : \mathbf{bool}\\ \pi \vdash p_1 \geq p_2 : \mathbf{bool}\end{array}} \qquad \frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{bool}\\ \pi \vdash p_2 : \mathbf{bool}\end{array}}{\begin{array}{l}\pi \vdash p_1 = p_2 : \mathbf{bool}\\ \pi \vdash p_1 \neq p_2 : \mathbf{bool}\end{array}}$$

- Boolean Expressions

$$\frac{\pi \vdash p : \mathbf{bool}}{\pi \vdash \sim p : \mathbf{bool}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{bool}\\ \pi \vdash p_2 : \mathbf{bool}\end{array}}{\begin{array}{l}\pi \vdash p_1 \wedge p_2 : \mathbf{bool}\\ \pi \vdash p_1 \vee p_2 : \mathbf{bool}\\ \pi \vdash p_1 \Rightarrow p_2 : \mathbf{bool}\\ \pi \vdash p_1 \Leftrightarrow p_2 : \mathbf{bool}\end{array}}$$

Here the only real novelty is the provision of two operators for exponentiation: $\uparrow$ accepts a real and an integer, and yields a real, while $**$ accepts two integers, and yields an integer (giving an error stop if its second operand is negative). The boolean operators $\Rightarrow$ and $\Leftrightarrow$ denote implication and equivalence (if-and-only-if) respectively.

As in Algol, the semicolon denotes sequential composition of commands. But now it can also be used to compose a command with a completion, giving a completion:

- Sequential Composition

$$\frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{comm}\\ \pi \vdash p_2 : \mathbf{comm}\end{array}}{\pi \vdash p_1 \,;\, p_2 : \mathbf{comm}} \qquad \frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{comm}\\ \pi \vdash p_2 : \mathbf{compl}\end{array}}{\pi \vdash p_1 \,;\, p_2 : \mathbf{compl}}$$

Two iterative constructions are provided: the traditional **while** command, and a **loop** construction which iterates its operand *ad infinitum* (i.e. until the operand jumps out of the loop by executing a completion):

- **while** Commands

$$\frac{\begin{array}{l}\pi \vdash p_1 : \mathbf{bool}\\ \pi \vdash p_2 : \mathbf{comm}\end{array}}{\pi \vdash \mathbf{while}\, p_1 \,\mathbf{do}\, p_2 : \mathbf{comm}}$$

- **loop** Completions

$$\frac{\pi \vdash p : \mathbf{comm}}{\pi \vdash \mathbf{loop}\, p : \mathbf{compl}}$$

There is also a phrase whose execution causes an error stop, with a message obtained by evaluating a character sequence:

- Error stops

$$\frac{\pi \vdash p : \mathbf{charseq}}{\pi \vdash \mathbf{error}\, p : \theta}$$

Notice that **error** p is a phrase that can have any type.

In place of the procedure declarations of Algol, Forsythe provides the more general **let**-definition construct invented by Peter Landin. In the implicitly typed case:

- Nonrecursive Definitions (with implicit typing)

$$\frac{\begin{array}{l}\pi \vdash p_1 : \theta_1 \\ \vdots \\ \pi \vdash p_n : \theta_n \\ [\,\pi \mid \iota_1 \colon \theta_1 \mid \cdots \mid \iota_n \colon \theta_n\,] \vdash p : \theta\end{array}}{\pi \vdash \mathbf{let}\,\iota_1 \equiv p_1, \ldots, \iota_n \equiv p_n \,\mathbf{in}\, p : \theta}$$

For example, in place of the Algol block

$$\mathbf{begin\ procedure}\ p(x);\ \theta\ x;\ B_{\text{proc}};\ B\ \mathbf{end}\ ,$$

one can write

$$\mathbf{let}\, p \equiv \lambda x\colon \theta.\ B_{\text{proc}}\,\mathbf{in}\, B\ .$$

Such definitions are not limited to procedures. One can write

$$\mathbf{let}\, x \equiv 3\,\mathbf{in}\, B\ ,$$

which will have exactly the same meaning as the phrase obtained from B by substituting 3 for x. Note, however, that this is not a variable declaration; x has the type **int** (the type of 3) and cannot be assigned to within B. Moreover, if y is an integer variable then

$$\mathbf{let}\, x \equiv y\,\mathbf{in}\, B$$

has the same meaning as the phrase obtained from B by substituting y for x, i.e. x is defined to be an alias of y.

Definitions can also be explicitly typed, indeeed one can mix implicit and explicit typing in the same **let**-construction. To describe this situation, we adopt the convention that, when an inference rule contains the notation $\{\cdots\}^?$, it stands for two rules, obtained (i) by deleting the notation, and (ii) by replacing it by the contents of the braces. (When the notation occurs n times, the rule stands for the 2^n rules obtained by taking all possible combinations.) Using this notation, we have a general rule that includes the previous one as a special case.

- Nonrecursive Definitions

$$\frac{\begin{array}{l}\pi \vdash p_1 : \theta_1 \\ \vdots \\ \pi \vdash p_n : \theta_n \\ [\,\pi \mid \iota_1 \colon \theta_1 \mid \cdots \mid \iota_n \colon \theta_n\,] \vdash p : \theta\end{array}}{\pi \vdash \mathbf{let}\,\iota_1\{\colon \theta_1\}^? \equiv p_1, \ldots, \iota_n\{\colon \theta_n\}^? \equiv p_n \,\mathbf{in}\, p : \theta}$$

Here, the explicit occurrence of $:\theta_i$ in the definition constrains the type θ_i to be used in an application of the rule, while the absence of such an occurrence allows θ_i to be chosen arbitrarily. (Notice that the alternative operator | cannot be used in **let** constructions, nor in the recursive definitions discussed below.)

For example, in place of the procedure definition displayed earlier, one can specify the type of the procedure in the definition (instead of specifying the type of the procedure argument in the abstraction):

$$\mathbf{let}\, p\colon \theta \to \mathbf{comm} \equiv \lambda x.\ B_{\text{proc}}\, \mathbf{in}\, B\,.$$

There is also an alternative form of the nonrecursive definition,

$$\mathbf{letinline}\, \iota_1\{:\theta_1\}^? \equiv p_1, \ldots, \iota_n\{:\theta_n\}^? \equiv p_n\, \mathbf{in}\, p\,,$$

that has the same typing behavior and semantics as the **let** form, but causes the procedures or other entities being defined to be compiled into inline code.

Recursion is provided in two ways. On the one hand, there is a fixed-point operator **rec:**

- Fixed Points

$$\frac{\pi \vdash p : \theta \to \theta}{\pi \vdash \mathbf{rec}\colon \theta.\ p : \theta}$$

Notice that explicit typing is required here.

On the other hand, there is a form of recursive definition:

- Recursive Definitions

$$\frac{\begin{array}{l}[\,\pi \mid \iota_1\colon\theta_1 \mid \cdots \mid \iota_n\colon\theta_n\,] \vdash p_1 : \theta_1 \\ \quad\vdots \\ [\,\pi \mid \iota_1\colon\theta_1 \mid \cdots \mid \iota_n\colon\theta_n\,] \vdash p_n : \theta_n \\ [\,\pi \mid \iota_1\colon\theta_1 \mid \cdots \mid \iota_n\colon\theta_n\,] \vdash p : \theta\end{array}}{\pi \vdash \mathbf{letrec}\, \iota_1\colon\theta_1, \ldots, \iota_n\colon\theta_n\, \mathbf{where}\, \iota_1 \equiv p_1, \ldots, \iota_n \equiv p_n\, \mathbf{in}\, p : \theta}$$

In this form, the recursively defined identifiers and their types must be listed before the definitions themselves, so that the reader (and compiler) knows that these identifiers have been rebound, and what their types are, before reading any of the p_i.

To keep the above rule simple, we have assumed that the two lists in a recursive definition define the identifiers ι_1, ..., ι_n in the same order. In fact, however, the order of the items in each of the lists is arbitrary. However, for both the recursive and nonrecursive definitions, ι_1, ..., ι_n must be distinct identifiers.

Next we consider a construction for intersecting or "merging" meanings. Suppose p_1 has type θ_1, p_2 has type θ_2, and $\theta_1 \sqcup \theta_2 \simeq \mathbf{ns}$, so that $[\![\theta_1 \mathbin{\&} \theta_2]\!] \approx [\![\theta_1]\!] \times [\![\theta_2]\!]$. One might hope to write p_1, p_2 to denote a meaning of type $\theta_1 \mathbin{\&} \theta_2$.

Unfortunately, this conflicts with the behavior of subtypes, since p_1 and p_2 might have types θ_1' and θ_2' such that $\theta_1' \leq \theta_1$ and $\theta_2' \leq \theta_2$ but $\theta_1' \sqcup \theta_2' \not\leq \mathbf{ns}$. For example, although $(a \equiv 3, b \equiv 4)$ and $b \equiv 5$ respectively have types $a\colon \mathbf{int}$ and $b\colon \mathbf{int}$, whose least upper bound is **ns**, the phrase

$$(a \equiv 3, b \equiv 4), b \equiv 5$$

would be ambiguous.

Our solution to this problem is to permit p_1, p_2 only when p_2 is an abstraction or an object construction, whose meaning then overwrites all components of the meaning of p_1 that have procedural types, or all object types with the same field name. The inference rules are:

- Merging

$$\frac{[\,\pi \mid \iota\colon \theta_i\,] \vdash p_2 : \theta'}{\pi \vdash (p_1, \lambda\iota\{\colon \theta_1 \mid \cdots \mid \theta_n\}^?.\ p_2) : \theta_i \rightarrow \theta'}$$

$$\frac{\pi \vdash p_1 : \rho}{\pi \vdash (p_1, \lambda\iota\{\colon \theta_1 \mid \cdots \mid \theta_n\}^?.\ p_2) : \rho}$$

$$\frac{\pi \vdash p_1 : (\iota_1\colon \theta)}{\pi \vdash (p_1, \lambda\iota\{\colon \theta_1 \mid \cdots \mid \theta_n\}^?.\ p_2) : (\iota_1\colon \theta)}$$

$$\frac{\pi \vdash p_2 : \theta}{\pi \vdash (p_1, \iota \equiv p_2) : (\iota\colon \theta)}$$

$$\frac{\pi \vdash p_1 : \rho}{\pi \vdash (p_1, \iota \equiv p_2) : \rho}$$

$$\frac{\pi \vdash p_1 : \theta \rightarrow \theta'}{\pi \vdash (p_1, \iota \equiv p_2) : \theta \rightarrow \theta'}$$

$$\frac{\pi \vdash p_1 : (\iota_1\colon \theta)}{\pi \vdash (p_1, \iota \equiv p_2) : (\iota_1\colon \theta)} \quad \text{when } \iota \neq \iota_1$$

Next, we introduce a construction for defining a sequence by giving a list of its elements:

- Sequences

$$\frac{\begin{array}{l}\pi \vdash p_0 : \theta \\ \quad\vdots \\ \pi \vdash p_{n-1} : \theta\end{array}}{\pi \vdash \mathbf{seq}(p_0, \ldots, p_{n-1}) : (\mathbf{int} \rightarrow \theta)\ \&\ \mathit{len}\colon \mathbf{int}} \quad \text{when } n \geq 1$$

The effect of this construction is that, if e is an integer expression with value k such that $0 \le k < n$, then

$$\mathbf{seq}(p_0, \ldots, p_{n-1})\, e$$

has the same meaning as p_k (and thus can be used in the role of a **case** construction). Moreover,

$$\mathbf{seq}(p_0, \ldots, p_{n-1}).\mathit{len}$$

is an integer expression with value n.

Finally, we introduce yet another form of definition, to permit the user to let identifiers stand for types. The types occurring in phrases are generalized to type expressions that can contain type identifiers, which are given meaning by the inference rule

- Type Definitions

$$\frac{\pi \vdash (p/\iota_1, \ldots, \iota_n \rightarrow \theta_{1,i_1}, \ldots, \theta_{n,i_n}) : \theta \qquad\qquad \text{where } 1 \le i_1 \le m_1, \ldots, n \le i_n \le m_n}{\pi \vdash (\mathbf{lettype}\, \iota_1 \equiv \theta_{1,1} \mid \cdots \mid \theta_{1,m_1},\ \ldots\ ,\ \iota_n \equiv \theta_{n,1} \mid \cdots \mid \theta_{n,m_n}\, \mathbf{in}\, p) : \theta}$$

where $(p/\iota_1, \ldots, \iota_n \rightarrow \theta_{1,i_1}, \ldots, \theta_{n,i_n})$ denotes the result of simultaneously substituting θ_{1,i_1}, $\ldots$, θ_{n,i_n} for the free occurrences (as type identifiers) of $\iota_1, \ldots, \iota_n$ in type expressions within p. (As with the definitions described earlier, $\iota_1, \ldots, \iota_n$ must be distinct identifiers.)

As a simple example,

$$\mathbf{lettype}\, t \equiv \mathbf{int}\, \mathbf{in}\, \lambda x{:}\, t.\ \lambda y{:}\, t.\ x \times y + 2$$

will have type **int** → **int** → **int**. Notice that this is a transparent, rather than opaque, form of type definition; e.g. within its scope, t is equivalent to **int**, rather than being an abstract type represented by integers (which would make the above example ill-typed).

Using the alternative operator in this construction provides another way to define procedures with multiple types. For example,

$$\mathbf{lettype}\, t \equiv \mathbf{int} \mid \mathbf{real}\, \mathbf{in}\, \lambda x{:}\, t.\ \lambda y{:}\, t.\ x \times y + 2$$

will have both type **int** → **int** → **int** and **real** → **real** → **real**. (The use of the alternative operator in type definitions was suggested by Benjamin Pierce [18].)

The same string of characters can be used as both an ordinary identifier and as a type identifier without interference. A change in its binding as an ordinary identifier has no effect on its meaning as a type identifier, and vice-versa.

6. Predefined Identifiers

In place of various constants, Forsythe provides predefined (ordinary) identifiers, which may be redefined by the user, but which take on standard types and meanings outside of these bindings. In describing these identifiers, we simply state the type of their unbound occurrences, e.g. we write $\mathit{true}\colon \mathbf{bool}$ as an abbreviation for the inference rule

$$\frac{}{\pi \vdash \mathit{true} : \mathbf{bool}} \quad \text{when } \mathit{true} \notin \operatorname{dom} \pi \,.$$

In the first place, there are the usual boolean constants, a **skip** command that leaves the state unchanged, and a standard phrase of type **ns**:

$$\mathit{true}\colon \mathbf{bool} \qquad\qquad \mathit{false}\colon \mathbf{bool}$$

$$\mathit{skip}\colon \mathbf{comm} \qquad\qquad \mathit{null}\colon \mathbf{ns} \;.$$

(Of course, there are many other nonsense phrases — phrases whose only types are equivalent to **ns** — which are all too easy to write, but *null* is the only such phrase that will not activate a warning message from the compiler. The point is that there are contexts in which *null* is sensible, for example as the denotation of an object with no fields.)

The remaining predefined identifiers denote built-in procedures. Four of these procedures serve to declare variables. For $\delta = \mathbf{int}$, **real**, **bool**, or **char**:

$$\begin{aligned}
&\mathit{new\delta var}\colon \delta \rightarrow \\
&\qquad \big((\delta\mathbf{var} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm} \\
&\qquad \;\&\; (\delta\mathbf{var} \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl} \\
&\qquad \;\&\; (\delta\mathbf{var} \rightarrow \mathbf{int}) \rightarrow \mathbf{int} \\
&\qquad \;\&\; (\delta\mathbf{var} \rightarrow \mathbf{real}) \rightarrow \mathbf{real} \\
&\qquad \;\&\; (\delta\mathbf{var} \rightarrow \mathbf{bool}) \rightarrow \mathbf{bool} \\
&\qquad \;\&\; (\delta\mathbf{var} \rightarrow \mathbf{char}) \rightarrow \mathbf{char}\big) \,.
\end{aligned}$$

The application $\mathit{new\delta var}\ \mathit{init}\ p$ causes a new δ variable to be added to the state of the computation; this variable is initialized to the value *init*, then the procedure p is applied to the variable, and finally the new variable is removed from the state of the computation when the call of p is completed (or when the execution of a nonlocal completion causes control to escape from p so that the new variable can no longer be assigned or evaluated.) Thus

$$\mathit{newintvar}\ \ \mathit{init}\ \ \lambda x.\ B$$

is equivalent to the Algol block

$$\mathbf{begin\ integer}\ x;\ \ x := \mathit{init};\ \ B\ \mathbf{end}\,.$$

The multiplicity of types of the $\mathit{new\delta var}$ procedures permits variables to be declared in completions and expressions as well as commands. Within expressions, however, locally declared variables, like any other variables, can be evaluated but not assigned.

Four analogous procedures are provided for declaring variable sequences:

$$\begin{aligned}
&\mathit{new\delta varseq}\colon \mathbf{int} \to (\mathbf{int} \to \delta) \to \\
&\quad \Big((\delta\mathbf{varseq} \to \mathbf{comm}) \to \mathbf{comm} \\
&\quad \& \, (\delta\mathbf{varseq} \to \mathbf{compl}) \to \mathbf{compl} \\
&\quad \& \, (\delta\mathbf{varseq} \to \mathbf{int}) \to \mathbf{int} \\
&\quad \& \, (\delta\mathbf{varseq} \to \mathbf{real}) \to \mathbf{real} \\
&\quad \& \, (\delta\mathbf{varseq} \to \mathbf{bool}) \to \mathbf{bool} \\
&\quad \& \, (\delta\mathbf{varseq} \to \mathbf{char}) \to \mathbf{char}\Big) .
\end{aligned}$$

The application $\mathit{new\delta varseq}\ l\ \mathit{init}\ p$ causes a new δ variable sequence of length l to be added to the state of the computation; the elements of this sequence are initialized to values obtained by applying the procedure *init*, and then the procedure p is applied to the sequence. Thus

$$\mathit{newintvarseq}\ \ l\ \ \mathit{init}\ \ \lambda x.\, B$$

is equivalent to the Algol block

```
begin integer array $x(0 : l - 1)$;
    begin integer $i$;
    for $i := 0$ to $l - 1$ do $x(i) := \mathit{init}(i)$
    end;
$B$
end .
```

In essence, this approach to the declaration of variables and sequences is a syntactic desugaring of the conventional form of declarations into the application of a procedure; procedures such as *newintvar init* or *newintvarseq l init* that are intended to be used this way are called *declarators.* The advantage of this view is that the user can define his own declarators or declarator-producing procedures. For example (as we will illustrate later), the user can define his own declarators for any kind of array for which he can program the index-mapping function.

Another declarator is provided for the declaration of completions that cause control to escape from a command. The procedure

$$\mathit{escape}\colon (\mathbf{compl} \to \mathbf{comm}) \to \mathbf{comm}$$

applies its parameter to a completion whose execution causes immediate termination of the application of *escape.* Thus

$$\mathit{escape}\ \ \lambda e.\, C$$

is equivalent to the Algol block

$$\mathbf{begin}\ C';\ \ e\colon \mathbf{end}\ ,$$

where C' is obtained from C by substituting **goto** e for e.

Facilities for input and output are also provided by declarators. For output, there is

$$\begin{aligned}&\mathit{newoutchannel}\colon \mathbf{charseq} \to\\ &\quad\Big((\mathbf{characc} \to \mathbf{comm}) \to \mathbf{comm}\\ &\quad\ \&\ (\mathbf{characc} \to \mathbf{compl}) \to \mathbf{compl}\Big)\,.\end{aligned}$$

The application *newoutchannel* s p opens the file named by s for output and applies the procedure p to an *output channel* c, which has type **characc**. Each time p assigns a character to c, the character is output to the file. When the call of p is completed (or when the execution of a nonlocal completion causes control to escape from p so that c can no longer be executed), the file is closed. (Our choice of automatic file-closure upon block exit is based on a belief that, in an Algol-like language, file buffers should obey the same stack discipline as variables.)

Since output channels are character acceptors, one might expect input channels to be character expressions, but this would violate the principle that expressions must not have side effects. Instead, an input channel has type $(\mathbf{characc}\ \&\ \mathit{eof}\colon \mathbf{compl}) \to \mathbf{comm}$, so that the declarator for input has type

$$\begin{aligned}&\mathit{newinchannel}\colon \mathbf{charseq} \to\\ &\quad\Big(\Big(((\mathbf{characc}\ \&\ \mathit{eof}\colon \mathbf{compl}) \to \mathbf{comm}) \to \mathbf{comm}\Big) \to \mathbf{comm}\\ &\quad\ \&\ \Big(((\mathbf{characc}\ \&\ \mathit{eof}\colon \mathbf{compl}) \to \mathbf{comm}) \to \mathbf{compl}\Big) \to \mathbf{compl}\Big)\,.\end{aligned}$$

The application *newinchannel* s p opens the file named by s for input and applies the procedure p to an input channel c. Each time p executes a call $c(a, \mathit{eof} \equiv k)$ the next character is read from the file and passed as an argument to the character acceptor a, unless an end-of-file has occurred, in which case the completion k is executed. When the call of p is completed (or when a nonlocal completion causes an escape from p), the file is closed.

Standard input and output are provided by channels that are named by predefined identifiers:

$$\mathit{std\_in}\colon (\mathbf{characc}\ \&\ \mathit{eof}\colon \mathbf{compl}) \to \mathbf{comm} \qquad\qquad \mathit{std\_out}\colon \mathbf{characc}\,.$$

Some obvious functional procedures are provided to convert between characters and their integer codes:

$$\mathit{char\_to\_code}\colon \mathbf{char} \to \mathbf{int} \qquad\qquad \mathit{code\_to\_char}\colon \mathbf{int} \to \mathbf{char}\,,$$

and to convert character sequences in decimal notation into integers and real numbers:

$$\mathit{charseq\_to\_int}\colon \mathbf{charseq} \to \mathbf{int} \qquad\qquad \mathit{charseq\_to\_real}\colon \mathbf{charseq} \to \mathbf{real}\,.$$

The last two procedures ignore nondigits, except for leading minus signs and (in the case of *charseq_to_real*) the first occurrence of a decimal point.

One might expect the procedure that converts integers into their decimal representations to have the type **int** → **charseq**, but this would be unsuitable since **charseq** is not a datatype. Instead, there is another declarator:

$$
\begin{aligned}
&\mathit{int\_to\_charseq}\colon \mathbf{int} \rightarrow\\
&\qquad\big((\mathbf{charseq} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}\\
&\qquad\&(\mathbf{charseq} \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{int}) \rightarrow \mathbf{int}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{real}) \rightarrow \mathbf{real}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{bool}) \rightarrow \mathbf{bool}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{char}) \rightarrow \mathbf{char}\big)\,.
\end{aligned}
$$

The application *int_to_charseq* n p converts the integer n to a character sequence giving its decimal representation, and applies the procedure p to this character sequence.

The conversion of real numbers to a decimal representation is considerably more complex, for several reasons: One must deal with both a fraction and exponent, the digit-length of the fraction must be specified, and there is no universally accepted notation. The conversion is implemented by the declarator

$$
\begin{aligned}
&\mathit{real\_to\_charseq}\colon \mathbf{real} \rightarrow \mathbf{int} \rightarrow\\
&\qquad\big((\mathbf{charseq} \rightarrow \mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}\\
&\qquad\&(\mathbf{charseq} \rightarrow \mathbf{int} \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{int} \rightarrow \mathbf{real}) \rightarrow \mathbf{real}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}) \rightarrow \mathbf{bool}\\
&\qquad\&\,(\mathbf{charseq} \rightarrow \mathbf{int} \rightarrow \mathbf{char}) \rightarrow \mathbf{char}\big)\,.
\end{aligned}
$$

Let r be a positive nonzero real number and $f \times 10^x$ be a closest approximation to r such that x is an integer, $0.1 \leq f < 1.0$, and f has a decimal representation containing d digits (to the right of the decimal point). Then *real_to_charseq* r d p applies the procedure p to the digit sequence representing f (excluding the decimal point) and the integer x. (Notice that x, as well as f, can depend upon d when r is slightly less than a power of ten.)

Clearly, if Forsythe grows beyond the experimental stage, it will be necessary for the predefined identifiers to provide richer capabilities than are described above. To do this in an "upward compatible" manner, one can obviously add new predefined identifiers. But the type system provides another, more interesting possibility: One can lower the type of an existing predefined identifier to a subtype of its original type, and give a new meaning to the identifier, providing the implicit coercion induced by the subtype relation maps the new meaning back into the old one.

For example, one might change the type of *newoutchannel* to

$$\begin{aligned}&\mathit{newoutchannel}\colon \mathbf{charseq} \to\\ &\quad\Big(((\mathbf{characc}\ \&\ \mathit{flush}\colon \mathbf{comm}) \to \mathbf{comm}) \to \mathbf{comm}\\ &\quad\ \&\ ((\mathbf{characc}\ \&\ \mathit{flush}\colon \mathbf{comm}) \to \mathbf{compl}) \to \mathbf{compl}\Big)\,.\end{aligned}$$

As before, the application *newoutchannel* $s\ p$ would open the file s and apply the procedure p to an output channel c, and each time that p assigned a character to c, the character would be output to the file. But now p could also execute the command $c.\mathit{flush}$, which might flush the output buffer.

7. Syntactic Sugar

Several abbreviations are provided to avoid repeating type information (or the absence thereof) when several identifiers range over the same type. In types,

$$\iota_1, \ldots, \iota_n\colon \theta \quad \text{abbreviates} \quad \iota_1\colon \theta\ \&\ \cdots\ \&\ \iota_n\colon \theta\,.$$

In abstractions

$$\begin{gathered}\lambda\iota_1, \ldots, \iota_n\colon \theta_1 \mid \cdots \mid \theta_k.\ p \quad \text{abbreviates}\\ \lambda\iota_1\colon \theta_1 \mid \cdots \mid \theta_k.\ \ldots\ \lambda\iota_n\colon \theta_1 \mid \cdots \mid \theta_k.\ p\end{gathered}$$

and

$$\lambda\iota_1, \ldots, \iota_n.\ p \quad \text{abbreviates} \quad \lambda\iota_1.\ \ldots\ \lambda\iota_n.\ p\,.$$

In recursive definitions

$$\iota_1, \ldots, \iota_n\colon \theta \quad \text{abbreviates} \quad \iota_1\colon \theta, \ldots, \iota_n\colon \theta\,.$$

In each of these cases, the identifiers $\iota_1, \ldots, \iota_n$ must be distinct.

Also, to permit a more Algol-like appearance,

$$p_1 := p_2 \quad \text{abbreviates} \quad p_1 p_2\,.$$

Finally, although we treated them as independent constructions in Section 5, the definitional forms can be regarded as abbreviations. First, recursive definitions can be desugared in terms of nonrecursive definitions and the fixed-point operator. For a single definition, one can give a straightforward desugaring:

$$\mathbf{letrec}\,\iota_1\colon \theta_1\ \mathbf{where}\ \iota_1 \equiv p_1\ \mathbf{in}\ p \quad \text{abbreviates} \quad \mathbf{let}\,\iota_1 \equiv \mathbf{rec}\colon \theta_1.\ \lambda\iota_1.\ p_1\ \mathbf{in}\ p\,.$$

However, a general rule that includes simultaneous recursion is more complex:

$$\begin{aligned}&\mathbf{letrec}\,\iota_1\colon \theta_1, \ldots, \iota_n\colon \theta_n\ \mathbf{where}\ \iota_1 \equiv p_1, \ldots, \iota_n \equiv p_n\ \mathbf{in}\ p \quad \text{abbreviates}\\ &\quad \mathbf{let}\,\iota \equiv \mathbf{rec}\colon (\iota_1\colon \theta_1\ \&\ \cdots\ \&\ \iota_n\colon \theta_n).\ \lambda\iota.\\ &\qquad \mathbf{let}\,\iota_1 \equiv \iota.\iota_1, \ldots, \iota_n \equiv \iota.\iota_n\ \mathbf{in}(\iota_1 \equiv p_1, \ldots, \iota_n \equiv p_n)\\ &\quad \mathbf{in}\ \mathbf{let}\,\iota_1 \equiv \iota.\iota_1, \ldots, \iota_n \equiv \iota.\iota_n\ \mathbf{in}\ p\,,\end{aligned}$$

where ι is any identifier not occurring in the **letrec** definition.

Second, nonrecursive definitions can be desugared in terms of abstractions and applications (following Landin):

$$\mathbf{let}\, \iota_1 \colon \theta_1 \equiv p_1,\ \ldots\ , \iota_n \colon \theta_n \equiv p_n \,\mathbf{in}\, p \quad \text{abbreviates} \quad (\lambda \iota_1 \colon \theta_1 .\ \cdots \lambda \iota_n \colon \theta_n .\ p) p_1 \cdots p_n \,.$$

This rule continues to make sense if, for some i, the types θ_i are omitted. But in this case the rightside will contain too little information to satisfy the typechecker, even though the leftside may be satisfactory.

8. Reduction Rules

As mentioned in the introduction, an operational way of describing Forsythe is to say that a program is a phrase of type **comm**, in an enriched typed lambda calculus, that is executed by first reducing the phrase to normal form (more precisely, to a possibly infinite or partial normal form) and then executing the normal form, which will be a program in the simple imperative language. Although we will not pursue this view in detail, it is useful to list some of the reduction rules, which preserve the meanings of programs and thus provide insight into their semantics. (We will ignore types in these rules, since they play no role in the process of reduction.)

First there is the lambda-calculus rule of β-reduction:

$$(\lambda \iota .\ p_1) p_2 \quad \Longrightarrow \quad (p_1 / \iota \rightarrow p_2)$$

where $(p_1 / \iota \rightarrow p_2)$ denotes the result of substituting p_2 for the free occurrences of ι (except as a type identifier or field name) in p_1.

Then there is a rule for selecting fields:

$$(\iota \equiv p).\iota \quad \Longrightarrow \quad p \,,$$

two rules for conditionals:

$$(\mathbf{if}\, p_1 \,\mathbf{then}\, p_2 \,\mathbf{else}\, p_3) p_4 \quad \Longrightarrow \quad \mathbf{if}\, p_1 \,\mathbf{then}\, p_2\, p_4 \,\mathbf{else}\, p_3\, p_4$$

$$(\mathbf{if}\, p_1 \,\mathbf{then}\, p_2 \,\mathbf{else}\, p_3).\iota \quad \Longrightarrow \quad \mathbf{if}\, p_1 \,\mathbf{then}\, p_2.\iota \,\mathbf{else}\, p_3.\iota \,,$$

a rule for nonrecursive definitions:

$$\mathbf{let}\, \iota_1 \equiv p_1,\ \ldots\ , \iota_n \equiv p_n \,\mathbf{in}\, p \quad \Longrightarrow \quad (p / \iota_1\ \ldots\ , \iota_n \rightarrow p_1,\ \ldots\ , p_n) \,,$$

and a rule for the fixed-point operator:

$$\mathbf{rec}\, p \quad \Longrightarrow \quad p(\mathbf{rec}\, p) \,.$$

In addition, there are a number of rules dealing with the merging operation:

$$(p_1, \lambda\iota.\ p_2)p_3 \quad\Longrightarrow\quad (\lambda\iota.\ p_2)p_3$$

$$\begin{aligned}
(p_1, \iota \equiv p_2)p_3 &\quad\Longrightarrow\quad p_1\, p_3 \\
(p_1, \lambda\iota.\ p_2).\iota' &\quad\Longrightarrow\quad p_1.\iota' \\
(p_1, \iota \equiv p_2).\iota &\quad\Longrightarrow\quad (\iota \equiv p_2).\iota \\
(p_1, \iota \equiv p_2).\iota' &\quad\Longrightarrow\quad p_1.\iota' \quad \text{when } \iota \neq \iota'\,.
\end{aligned}$$

(It should be noted that these rules are not complete; in particular, it is not clear how to provide rules for reducing merges in contexts that require primitive types.)

The reduction rules make it clear that call by name pervades Forsythe. For example, if p_c is any phrase that does not contain free occurrences of ι, and p_1 and p_2 are any phrases, then

$$\begin{aligned}
(\lambda\iota.\ p_c)p_1 &\quad\Longrightarrow\quad p_c \\
\mathbf{let}\,\iota \equiv p_1\,\mathbf{in}\,p_c &\quad\Longrightarrow\quad p_c \\
(\iota_1 \equiv p_1, \iota_2 \equiv p_2).\iota_1 &\quad\Longrightarrow\quad p_1 \\
(\iota_1 \equiv p_1, \iota_2 \equiv p_2).\iota_2 &\quad\Longrightarrow\quad p_2
\end{aligned}$$

hold even when p_1 or p_2 denote nonterminating computations.

Morever, call by name even characterizes the assignment operation, since assignments are abbreviations for procedure calls. For example, assuming that x is an ordinary integer variable (e.g. declared using *newintvar*),

$$(\lambda y.\ x := 3) := p \qquad \text{and} \qquad (\lambda y.\ x := y + y) := p$$

would evaluate the expression p zero and two times respectively. This is probably the most controversial design decision in Forsythe, since it makes the language, so to speak, more Algol-like than Algol itself. It may degrade the efficiency with which the language can be implemented but, as demonstrated in Sections 11 and 13, it leads to some interesting programming techniques.

9. Examples of Procedures

In this and the next four sections, we provide a variety of examples of Forsythe programs. Many of these examples are translations of Algol W programs given in [10], which the reader may wish to compare with the present versions.

To define a proper procedure that sets its second parameter to the factorial of its first parameter, we define *fact* to be the obvious command, abstracted on an integer expression

n and an integer variable f:

```
let fact: int → intvar → comm ≡ λn. λf.
    newintvar 0 λk.
        (f := 1 ; while k ≠ n do (k := k + 1 ; f := k × f))
```

Here we have specified the necessary types in the nonrecursive definition, but instead we could have specified them in the abstractions:

```
let fact ≡ λn: int. λf: intvar.
    newintvar 0 λk.
        (f := 1 ; while k ≠ n do (k := k + 1 ; f := k × f))
```

(Although Forsythe supports either method of specifying the types of nonrecursive procedures, in these examples we will usually give types in definitions rather than in abstractions, since this approach is more readable, and in some cases gives more efficient typechecking.)

This procedure has the usual shortcoming of call by name: It will repeatedly evaluate the expression n. To remedy this defect, we replace n by a local variable (also called n) that is initialized to the input parameter n. Notice that this is equivalent to the definition of call by value in Algol 60.

```
let fact: int → intvar → comm ≡ λn. λf.
    newintvar n λn.
        newintvar 0 λk.
            (f := 1 ; while k ≠ n do (k := k + 1 ; f := k × f))
```

We can also modify this procedure to obtain the effect of calling f by result (as in Algol W [4]). We replace f by a local variable, and then assign the final value of this local variable to the parameter f, which now has type **intacc**, since it is never evaluated by the procedure.

```
let fact: int → intacc → comm ≡ λn. λf.
    newintvar n λn. newintvar 1 λlocalf.
        (newintvar 0 λk.
            while k ≠ n do (k := k + 1 ; localf := k × localf);
        f := localf)
```

This transformation is sufficiently complex that it is worthwhile to encapsulate it as a procedure. We define

```
letinline newintvarres: int → intacc → (intvar → comm) → comm ≡
    λinit. λfin. λb. newintvar init λlocal. (b local ; fin := local)
```

Then to call f by result we define

```
let fact: int → intacc → comm ≡ λn. λf.
    newintvar n λn. newintvarres 1 f λf.
        newintvar 0 λk.
            while k ≠ n do (k := k + 1 ; f := k × f)
```

When placed within the scope of the definition of *newintvarres*, this definition of *fact* reduces to to the previous one (except for the names of bound identifiers) and therefore has the same meaning. Moreover, since *newintvarres* is defined by **letinline**, the two definitions of *fact* will compile into the same machine code.

We can also define the traditional recursive function procedure for computing the factorial. Here again we call n by value, illustrating the use of *newintvar* within an expression.

```
letrec fact: int → int
where fact ≡ λn. newintvar n λn.
    if n = 0 then 1 else n × fact(n − 1)
```

Next, we give some examples of procedures that take advantage of call by name. In the following function procedure for integer multiplication, call by name is used to provide "short-circuit" evaluation,

```
letinline multiply: int → int → int ≡ λm. λn. if m = 0 then 0 else m × n
```

i.e. n will not be evaluated when m is zero. In a proper procedure akin to the Pascal **repeat** command,

```
letinline repeat: comm → bool → comm ≡ λc. λb. (c ; while ~b do c)
```

b must be called by name to permit its repeated evaluation. (Both *multiply* and *repeat* are such simple procedures that it is obviously worthwhile to compile them inline.)

Repeated evaluation is also crucial to the following program, where the call of the procedure *sum* sets s to $\sum_{i=a}^{b} X(i) \times Y(i)$ by repeatedly evaluating $X(i) \times Y(i)$ while increasing the variable i:

```
letinline sum: intvar → int → comm ≡ λi. λe.
    begin s := 0 ; i := a − 1;
    while i < b do (i := i + 1 ; s := s + e)
    end
in sum i (X(i) × Y(i))
```

This way of using call by name, known as "Jensen's device", was illustrated in the original Algol 60 Report [2, 3] by the exemplary procedure *Innerproduct.*

Finally, we give two higher-order procedures akin to the **for** command:

```
**letinline** $for$: $\mathbf{int} \to \mathbf{int} \to (\mathbf{int} \to \mathbf{comm}) \to \mathbf{comm} \equiv \lambda l.\ \lambda u.\ \lambda b.$
    $newintvar(l - 1)\ \lambda k.\ newintvar\ u\ \lambda u.$
        $\mathbf{while}\ k < u\ \mathbf{do}\ (k := k + 1\,;\, b\ k),$
$fordown$: $\mathbf{int} \to \mathbf{int} \to (\mathbf{int} \to \mathbf{comm}) \to \mathbf{comm} \equiv \lambda l.\ \lambda u.\ \lambda b.$
    $newintvar(u + 1)\ \lambda k.\ newintvar\ l\ \lambda l.$
        $\mathbf{while}\ k > l\ \mathbf{do}\ (k := k - 1\,;\, b\ k)$
**in** $for\ 0\ 9\ \lambda i.\ s := s + X(i) \times Y(i)$
```

Notice that, in these procedures, since the procedure b takes a parameter of type **int**, the application $b\ k$ cannot change the value of k. Moreover, although this application can change the values of the parameters l and u, the interval iterated over is always determined by the initial values of these parameters.

Even though the procedures *sum*, *for*, and *fordown* are moderately complex, we have used **letinline** to define them, since they evaluate some of their parameters repeatedly. When a procedure is defined by **letinline**, not only are its calls compiled into inline code, but also the execution or call of the parameters of the procedure. In particular, the expressions i and $X(i) \times Y(i)$ in the call of *sum* will be executed inline, and the procedure $\lambda i.\ s := s + X(i) \times Y(i)$ in the call of *for* will be called inline.

10. Escapes and Completions

The procedure *escape* declares a completion whose execution causes an exit from the call of *escape*. A simple example of its use is the following procedure for searching an integer function X (which might be an integer sequence or array) over the interval l to u for a value that is equal to y. If such a value is found, the procedure sets *present* to *true* and j to the argument for which $X(j) = y$; otherwise it sets *present* to *false*.

```
**let** $linsearch$: $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \to \mathbf{int} \to \mathbf{boolacc} \to \mathbf{intacc} \to \mathbf{comm} \equiv$
    $\lambda X.\ \lambda l.\ \lambda u.\ \lambda y.\ \lambda present.\ \lambda j.$
        $escape\ \lambda out.$
            $\big(for\ l\ u\ \lambda k.\ \mathbf{if}\ X(k) = y\ \mathbf{then}\ (present := true\,;\, j := k\,;\, out)\ \mathbf{else}\ skip;$
            $present := false\big)$
```

An alternative version of this procedure branches to one of two parameters depending upon whether the search succeeds. If the search fails, it goes to the completion *failure*; if the search succeeds, it goes to the completion procedure *success*, passing it the integer k such that $X(k) = y$.

```
$\mathbf{let}\ \mathit{linsearch}$:
    $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \to \mathbf{int} \to (\mathbf{int} \to \mathbf{compl}) \to \mathbf{compl} \to \mathbf{compl} \equiv$
        $\lambda X.\ \lambda l.\ \lambda u.\ \lambda y.\ \lambda \mathit{success}.\ \lambda \mathit{failure}.$
            $\big(\mathit{for}\ l\ u\ \lambda k.\ \mathbf{if}\ X(k) = y\ \mathbf{then}\ \mathit{success}\ k\ \mathbf{else}\ \mathit{skip}\ ; \mathit{failure}\big)$
```

This illustrates that, if a procedure always terminates by executing a completion, then a call of the procedure will itself be a completion. This fact is relevant when the last action of a procedure is to assign to a parameter, since an assignment to a parameter is syntactic sugar for a call of the parameter. For example, the following is an alternative typing of the procedure *newintvarres* introduced in the previous section:

```
$\mathbf{letinline}\ \mathit{newintvarres}\colon \mathbf{int} \to (\mathbf{int} \to \mathbf{compl}) \to (\mathbf{intvar} \to \mathbf{comm}) \to \mathbf{compl} \equiv$
    $\lambda \mathit{init}.\ \lambda \mathit{fin}.\ \lambda b.\ \mathit{newintvar}\ \mathit{init}\ \lambda \mathit{local}.\ (b\ \mathit{local}\ ; \mathit{fin} := \mathit{local})$
```

This makes sense because the final assignment *fin* := *local*, really means *fin local*, which will be a completion when *fin* has type **int** → **compl**. One can even use intersection to give *newintvarres* both its conventional type and this variant in a single definition:

```
$\mathbf{letinline}\ \mathit{newintvarres}\colon \mathbf{int} \to \big(\mathbf{intacc} \to (\mathbf{intvar} \to \mathbf{comm}) \to \mathbf{comm}$
        $\&\ (\mathbf{int} \to \mathbf{compl}) \to (\mathbf{intvar} \to \mathbf{comm}) \to \mathbf{compl}\big) \equiv$
    $\lambda \mathit{init}.\ \lambda \mathit{fin}.\ \lambda b.\ \mathit{newintvar}\ \mathit{init}\ \lambda \mathit{local}.\ (b\ \mathit{local}\ ; \mathit{fin} := \mathit{local})$
```

In addition to using *escape*, one can define completions recursively, to obtain the equivalent of conventional labels. For example, the following procedure sets y to x^n (in time $\log n$), without doing unnecessary tests:

```
$\mathbf{let}\ \mathit{power}\colon \mathbf{int} \to \mathbf{int} \to \mathbf{intacc} \to \mathbf{comm} \equiv \lambda x.\ \lambda n.\ \lambda y.$
    $\mathit{newintvar}\ n\ \lambda k.\ \mathit{newintvarres}\ 1\ y\ \lambda y.\ \mathit{newintvar}\ x\ \lambda z.$
        $\mathit{escape}\ \lambda \mathit{zr}\ \{k = \mathit{0}\}.$
            $\mathbf{letrec}\ \mathit{tr}, \mathit{nz}, \mathit{ev}, \mathit{od}, \mathit{nzev}\colon \mathbf{compl}$
            $\mathbf{where}$
                $\mathit{tr}\ \{\mathit{true}\} \equiv \mathbf{if}\ k = 0\ \mathbf{then}\ \mathit{zr}\ \mathbf{else}\ \mathit{nz},$
                $\mathit{nz}\ \{k \neq 0\} \equiv \mathbf{if}\ k\,\mathbf{rem}\,2 \neq 0\ \mathbf{then}\ \mathit{od}\ \mathbf{else}\ \mathit{nzev},$
                $\mathit{ev}\ \{\mathit{even}\ k\} \equiv \mathbf{if}\ k = 0\ \mathbf{then}\ \mathit{zr}\ \mathbf{else}\ \mathit{nzev},$
                $\mathit{od}\ \{\mathit{odd}\ k\} \equiv k := k - 1\ ; y := y \times z\ ; \mathit{ev},$
                $\mathit{nzev}\ \{k \neq 0 \wedge \mathit{even}\ k\} \equiv k := k \div 2\ ; z := z \times z\ ; \mathit{nz}$
            $\mathbf{in}\ \mathit{tr}$
```

The invariant of this program, i.e. the assertion that holds whenever any completion is executed, is

$$y \times z^k = x^n \wedge k \geq 0\,.$$

The additional assertions given as comments at the binding of each completion hold whenever the corresponding completion is executed.

Notice that, in contrast to labels, one can never execute a completion by "passing through" to its definition. Indeed, the meaning of the above program is independent of the order of the definitions of completions.

11. Sequences and Arrays

As we remarked in Section 6, using the built-in procedures for declaring variable sequences the programmer can define his own procedures for declaring more complex kinds of arrays. For example, suppose we want Algol-like one-dimensional integer arrays with arbitrary lower and upper bounds (denoted by the field names *ll* and *ul*). First we define abbreviations for the relevant types:

$$\begin{aligned}
&\textbf{lettype intarray} \equiv (\textbf{int} \rightarrow \textbf{int} \;\&\; ll, ul\colon \textbf{int}),\\
&\qquad \textbf{intaccarray} \equiv (\textbf{int} \rightarrow \textbf{intacc} \;\&\; ll, ul\colon \textbf{int}),\\
&\qquad \textbf{intvararray} \equiv (\textbf{int} \rightarrow \textbf{intvar} \;\&\; ll, ul\colon \textbf{int})
\end{aligned}$$

Then the following procedure serves to declare integer variable arrays, which are defined in terms of integer sequences:

$$\begin{aligned}
&\textbf{letinline}\ \mathit{newintvararray}\colon \textbf{int} \rightarrow \textbf{int} \rightarrow (\textbf{int} \rightarrow \textbf{int}) \rightarrow\\
&\qquad \Big((\textbf{intvararray} \rightarrow \textbf{comm}) \rightarrow \textbf{comm} \;\&\; (\textbf{intvararray} \rightarrow \textbf{compl}) \rightarrow \textbf{compl}\\
&\qquad \;\&\; (\textbf{intvararray} \rightarrow \textbf{int}) \rightarrow \textbf{int} \;\&\; (\textbf{intvararray} \rightarrow \textbf{real}) \rightarrow \textbf{real}\\
&\qquad \;\&\; (\textbf{intvararray} \rightarrow \textbf{bool}) \rightarrow \textbf{bool} \;\&\; (\textbf{intvararray} \rightarrow \textbf{char}) \rightarrow \textbf{char}\Big) \equiv\\
&\qquad\qquad \lambda l.\ \lambda u.\ \lambda \mathit{init}.\ \lambda b.\ \mathit{newintvar}\ l\ \lambda l.\ \mathit{newintvar}\ u\ \lambda u.\\
&\qquad\qquad\qquad \mathit{newintvarseq}(u - l + 1)\big(\lambda k.\ \mathit{init}(k + l)\big)\lambda X.\\
&\qquad\qquad\qquad\quad b\big(\lambda k.\ X(k - l), ll \equiv l, ul \equiv u\big)
\end{aligned}$$

The sixfold intersection that is the type of *newintvararray* is similar to the type of built-in declarators such as *newintvarseq*. Equally well, one could provide the necessary type information in the abstractions rather than the definition, using the alternative operator:

$$\begin{aligned}
&\textbf{letinline}\ \mathit{newintvararray} \equiv \lambda l, u\colon \textbf{int}.\ \lambda \mathit{init}\colon \textbf{int} \rightarrow \textbf{int}.\\
&\qquad \lambda b\colon \textbf{intvararray} \rightarrow \textbf{comm} \mid \textbf{intvararray} \rightarrow \textbf{compl} \mid \textbf{intvararray} \rightarrow \textbf{int} \mid\\
&\qquad\qquad \textbf{intvararray} \rightarrow \textbf{real} \mid \textbf{intvararray} \rightarrow \textbf{bool} \mid \textbf{intvararray} \rightarrow \textbf{char}.
\end{aligned}$$

```
$\mathit{newintvar}\ l\ \lambda l.\ \mathit{newintvar}\ u\ \lambda u.$
    $\mathit{newintvarseq}(u - l + 1)\big(\lambda k.\ \mathit{init}(k + l)\big)\lambda X.$
        $b\big(\lambda k.\ X(k - l), ll \equiv l, ul \equiv u\big)$
```

We can also define a procedure *slice* that, given an array and two integers, yields a subsegment of the array with new bounds. The simplest definition is

```
$\mathbf{letinline}\ \mathit{slice}\colon \big((\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \to \mathbf{intarray}$
    $\&\ (\mathbf{int} \to \mathbf{intacc}) \to \mathbf{int} \to \mathbf{int} \to \mathbf{intaccarray}\big) \equiv$
        $\lambda X.\ \lambda l.\ \lambda u.\ (X, ll \equiv l, ul \equiv u)$
```

A safer alternative, which checks applications of the array against the new bounds, is

```
$\mathbf{letinline}\ \mathit{slicecheck}\colon \big((\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \to \mathbf{intarray}$
    $\&\ (\mathbf{int} \to \mathbf{intacc}) \to \mathbf{int} \to \mathbf{int} \to \mathbf{intaccarray}\big) \equiv$
        $\lambda X.\ \lambda l.\ \lambda u.\ (\lambda k.\ \mathbf{if}\ l \leq k \wedge k \leq u\ \mathbf{then}\ X\ k\ \mathbf{else}\ \mathbf{error}\ \text{`subscript error'},$
            $ll \equiv l, ul \equiv u)$
```

The type of *slice* and *slicecheck* is extremely general:

- If the first argument has type **int** → **int**, or any subtype, such as **intseq** or **intarray**, then the call will have type **intarray**.
- If the first argument has type **int** → **intacc**, or any subtype, such as **intaccseq** or **intaccarray**, then the call will have type **intaccarray**.
- If the first argument has type **int** → **intvar**, or any subtype, such as **intvarseq** or **intvararray**, then both of the previous cases will apply, and the call will have type **intarray** & **intaccarray**, which is equivalent to **intvararray**.

The use of these procedures is illustrated by a program for sorting by finding maxima. First we define a procedure that sets j to the subscript of a maximum of an array X:

```
$\mathbf{letinline}\ \mathit{max}\colon \mathbf{intarray} \to \mathbf{intacc} \to \mathbf{comm} \equiv \lambda X.\ \lambda j.$
    $\mathit{newintvar}\ X.ll\ \lambda a.\ \mathit{newintvar}\ X.ul\ \lambda b.$
        $\mathit{newintvarres}\ a\ j\ \lambda j.$
            $\mathbf{while}\ a < b\ \mathbf{do}\ (a := a + 1\, ;\mathbf{if}\ X\ a > X\ j\ \mathbf{then}\ j := a\ \mathbf{else}\ \mathit{skip})$
```

(Here giving X the type **intarray** rather than **intvararray** indicates that *max* will examine X but not assign to it.) Next comes a procedure for exchanging a pair of array elements:

```
letinline exchange: (int → intvar) → int → int → comm ≡ λX. λi. λj.
    newintvar i λi. newintvar j λj.
        newintvar(X i) λt. (X i := X j ; X j := t)
```

(Giving X the type **int** → **intvar** indicates that *exchange* does not evaluate bounds; e.g. it would also be applicable to an integer variable sequence.) Then the sort procedure is:

```
let maxsort: intvararray → comm ≡ λX.
    newintvar X.ll λa. newintvar X.ul λb.
        while a ≤ b do newintvar 0 λj.
            (max(slice X a b) j ; exchange X j b ; b := b − 1)
```

The above procedure contains a spurious initialization of the variable j. The purpose of this variable is to accept the result of *max*, but *newintvar* requires us to initialize it before calling *max*. However, this unpleasantness can be avoided by taking advantage of the fact that assignments are really applications. By substituting the definition of *newintvarres* into the definition of *max* and reducing, we find that the definition of *max* is equivalent to

```
letinline max: intarray → intacc → comm ≡ λX. λj.
    newintvar X.ll λa. newintvar X.ul λb.
        newintvar a λlocal.
            (while a < b do
                (a := a + 1 ; if X a > X local then local := a else skip);
            j := local)
```

where $j :=$ *local* is syntactic sugar for j *local*. Thus the second parameter of *max* can be any procedure of type **int** → **comm**, i.e. any proper procedure accepting an integer; the effect of *max* will be to apply this procedure to the subscript of the maximum of X.

To avoid the spurious initialization, we make this parameter a procedure that carries out the appropriate exchange, dispensing with the variable j entirely:

```
let maxsort: intvararray → comm ≡ λX.
    newintvar X.ll λa. newintvar X.ul λb.
        while a ≤ b do
            (max(slice X a b) λj. exchange X j b ; b := b − 1)
```

A similar use of "generalized call by result" occurs in the following definition of quicksort:

```
letinline partition: intvararray → int → intacc → comm ≡ $\lambda X.\ \lambda r.\ \lambda p.$
    newintvar $X.ll$ $\lambda c.$ newintvar $X.ul$ $\lambda d.$ newintvar $r$ $\lambda r.$
        (while $c \leq d$ do
            if $X\ c \leq r$ then $c := c + 1$ else
            if $X\ d > r$ then $d := d - 1$ else
                (exchange $X\ c\ d$ ; $c := c + 1$ ; $d := d - 1$);
        $p\ c$)
in
letrec quicksort: intvararray → comm
where quicksort ≡ $\lambda X.$
    newintvar $X.ll$ $\lambda a.$ newintvar $X.ul$ $\lambda b.$
        if $a < b$ then
            if $X\ a > X\ b$ then exchange $X\ a\ b$ else skip;
            partition(slice $X\ (a+1)\ (b-1)$) $((X\ a + X\ b) \div 2)$ $\lambda c.$
                (quicksort(slice $X\ a\ (c-1)$) ; quicksort(slice $X\ c\ b$))
        else skip
```

As a further example of the power of declarators, we can define the type of triangular arrays of real variables, along with an appropriate declarator (which, to keep the example simple, initializes the array elements to zero):

```
lettype trivararray ≡ ((int → int → realvar) & size: int) in
letinline newtrivararray: int →
    ((trivararray → comm) → comm & (trivararray → compl) → compl
    & (trivararray → int) → int & (trivararray → real) → real
    & (trivararray → bool) → bool & (trivararray → char) → char) ≡
        $\lambda n.\ \lambda b.$ newintvar $n$ $\lambda n.$ newrealvarseq$((n \times (n+1)) \div 2)(\lambda k.\ 0)\ \lambda X.$
            $b(\lambda i.\ \lambda j.$
                if $0 \leq j \wedge j \leq i \wedge i < n$ then $X((i \times (i+1)) \div 2 + j)$
                    else error 'subscript error',
                size ≡ $n)$ .
```

12. Input and Output

The following program illustrates the facilities for input and output. It reads a sequence of pairs of nonnegative integers from a file named 'infile' and writes the real quotient of each pair in floating-point notation to a file named 'outfile'. It is assumed that the integers in the input file are separated by sequences of one or more nondigits; if there are an odd number integers, the last is simply converted from integer to real.

Each number in the output is printed on a separate line, as a six-digit number greater or equal to one and less than ten, times a power of ten. A result of zero is printed as 0.0, while division by zero gives an error message.

```
newinchannel 'infile' λic. newoutchannel 'outfile' λoc.
letinline is_digit: char → bool ≡ λc. newcharvar c λc. #0 ≤ c ∧ c ≤ #9,
    writecharseq: charseq → comm ≡ λs. newintvar 0 λi.
        while i < s.len do (oc := s i ; i := i + 1)
in
letinline writereal: real → comm ≡ λr.
        if r = 0 then writecharseq '0.0' else
            real_to_charseq r 6 λs. λx.
                (oc := s 0 ; oc := #. ; writecharseq(λi. s(i + 1), len ≡ 5) ;
                writecharseq '*10**' ; int_to_charseq (x − 1) writecharseq),
    readint: compl → intacc → comm ≡ λnonumber. λa.
        newcharvarseq 30 (λk. #0) λs. newintvar 0 λi.
            (repeat (ic(s i, eof ≡ nonumber)) (is_digit(s i)) ;
                escape λe.
                    repeat (i := i + 1 ; ic(s i, eof ≡ e)) (∼ is_digit(s i)) ;
                a := charseq_to_int(s, len ≡ i))
in
escape λdone. loop
    readint done λm.
        readint (writereal m ; oc := #\n ; done) λn.
            if n = 0 then writecharseq 'division by zero\n' else
                (writereal(m/n) ; oc := #\n) .
```

Here *repeat* refers to the procedure defined in Section 9, while #0, #9, #., and #\n are character constants denoting the digits 0 and 9, the decimal point, and the new-line character.

The procedure *readint* uses a local character variable sequence to store the digit sequence being read, so that the number of digits is limited (to 29 in this example) by the length of the sequence. In fact, it is possible to avoid this limitation by programming *readint* recursively and using character sequences that are procedural functions rather than values of a variable sequence:

```
letrec readint: compl → intacc → comm, readint1: charseq → intacc → comm
where
    readint ≡ λnonumber. λa.
        ic(λc. newcharvar c λc.
            if is_digit c then readint1 (λi. c, len ≡ 1) a else
                readint nonumber a,
        eof ≡ nonumber),
    readint1 ≡ λs. λa. newintvar s.len λl. escape λe.
        ic(λc. newcharvar c λc.
            if is_digit c then readint1 (λi. if i = l then c else s i, len ≡ l + 1) a else
                a := charseq_to_int s,
        eof ≡ a := charseq_to_int s ; e) .
```

Here *readint1 s a* reads digits until encountering a nondigit, appends these digits on the right of the sequence s, converts the resulting sequence into an integer, and assigns the integer to a.

Unfortunately, however, this version of *readint* is neither perspicuous nor efficient — and is not recommended as good programming style.

13. Data Abstraction with Objects

Perhaps the most important way in which Forsythe is more general than Algol is in its provision of objects, which are a powerful tool for data abstraction. One can write abstract programs in which various kinds of data are realized by types of objects, and then encapsulate the representation of the data, and the expression of primitive operations in terms of this representation, in declarators for the objects.

To illustrate this style of programming, we will develop a program for computing reachability in a finite directed graph. Specifically, we will define a procedure *reachable* that, given a node x and a graph g, will compute the set of nodes that can be reached from x.

Throughout most of this development we will assume that "**node**" is a new data type; eventually we will see how this assumption can be eliminated. Given **node**, we can define a "**set**" to be an object denoting a finite set of nodes, whose fields (called methods in the jargon of object-oriented programming) are procedures for manipulating the denoted set:

```
$\textbf{lettype set} \equiv$
    $\big(\mathit{member}\colon \textbf{node} \rightarrow \textbf{bool}$
    $\&\ \mathit{insertnew}\colon \textbf{node} \rightarrow \textbf{comm}$
    $\&\ \mathit{iter}\colon (\textbf{node} \rightarrow \textbf{comm}) \rightarrow \textbf{comm}$
    $\&\ \mathit{pick}\colon \textbf{comm} \rightarrow (\textbf{node} \rightarrow \textbf{comm}) \rightarrow \textbf{comm}\big)$
```

The intention is that, if s is a set, x is a node, d is a procedure of type **node → comm**, and e is a command, then:

- $s.\,\mathit{member}\ x$ gives *true* if and only if $x \in s$.
- $s.\,\mathit{insertnew}\ x$ inserts x into s, providing x is not already in s.
- $s.\,\mathit{iter}\ d$ applies d to each member of s.
- If s is empty then $s.\,\mathit{pick}\ e\,d$ executes e; otherwise $s.\,\mathit{pick}\ e\,d$ removes an arbitrary member from s and applies d to the removed member.

In terms of **set**, we can give a naive version of the reachability procedure. The procedure maintains a set t of all nodes that have been found to be reachable from x, and a set u of those members of t whose immediate successors have yet to be added to t. (An immediate successor of a node y is a node that can be reached from y in one step.) Thus its invariant is

$$x \in t \wedge u \subseteq t \wedge (\forall y \in t)\ y \text{ is reachable from } x \wedge (\forall y \in t - u)\ g\,y \subseteq t\,,$$

where g is a function of type **node → set** such that $g\,y$ is the set of immediate successors of y. This invariant implies that, when u is empty, t is the set of all nodes reachable from x.

In writing *reachable*, we assume that the parameter g is the immediate-successor function of the graph, and that the result is to be communicated by applying a procedural parameter p to the final value of t:

```
$\textbf{let}\ \mathit{reachable}\colon \textbf{node} \rightarrow (\textbf{node} \rightarrow \textbf{set}) \rightarrow (\textbf{set} \rightarrow \textbf{comm}) \rightarrow \textbf{comm} \equiv$
    $\lambda x.\ \lambda g.\ \lambda p.\ \mathit{newset}\ \lambda t.\ \mathit{newset}\ \lambda u.$
        $\big(t.\mathit{insertnew}\ x\ ;\ u.\mathit{insertnew}\ x\ ;$
        $\mathit{escape}\ \lambda \mathit{out}.\ \textbf{loop}\ u.\mathit{pick}\ \mathit{out}\ \lambda y.\ (g\,y).\mathit{iter}\ \lambda z.$
            $\textbf{if}\ \sim t.\mathit{member}\ z\ \textbf{then}\ (t.\mathit{insertnew}\ z\ ;\ u.\mathit{insertnew}\ z)$
                $\textbf{else}\ \mathit{skip}\ ;$
        $p\,t\big)$
```

Here *newset* is a declarator that creates an object of type **set**, initialized to the empty set. Thus

$$\mathit{newset} : (\textbf{set} \rightarrow \textbf{comm}) \rightarrow \textbf{comm}\,.$$

Actually, we could give *newset* the more general type

$$(\mathbf{set} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm} \,\&\, (\mathbf{set} \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl}\,,$$

which would allow *reachable* to have the more general type

$$\mathbf{node} \rightarrow (\mathbf{node} \rightarrow \mathbf{set}) \rightarrow \Big((\mathbf{set} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm} \,\&\, (\mathbf{set} \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl}\Big)\,.$$

This generality, however, is unnecessary for our example and would distract from our argument. Thus, in this section, we will limit our declarators to the case where their calls are commands.

Next, we refine the reachability procedure to provide greater flexibility for the representation of sets. In place of the object type **set**, we introduce different object types for the different sets used in the program:

- **setg** for the sets produced by applying g,
- **sett** for the set t,
- **setu** for the set u.

The basic idea is to limit the fields of each of these object types to those procedures that are actually needed by our program. However, even greater flexibility is gained by taking advantage of the fact that the sets t and u are declared at the same time, and that u is always a subset of t. For this purpose, we introduce a "double declarator",

$$\mathit{newdoubleset} : (\mathbf{sett} \rightarrow \mathbf{setu} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$$

such that *newdoubleset* $\lambda t\colon \mathbf{sett}.\ \lambda u\colon \mathbf{setu}.\ C$ executes C after binding both t and u to new (initially empty) sets. Morever, to enforce the invariant $u \subseteq t$, we will eliminate the operation $t.\,\mathit{insertnew}$ and redefine $u.\,\mathit{insertnew}$ to insert its argument (which must not already belong to t) into both u and t.

Thus we have

```
**lettype setg** ≡ (iter: (node → comm) → comm),
        **sett** ≡ (member: node → bool & iter: (node → comm) → comm),
        **setu** ≡ (insertnew: node → comm
                & pick: comm → (node → comm) → comm)
in
let reachable: node → (node → setg) → (sett → comm) → comm ≡
        λx. λg. λp. newdoubleset λt. λu.
            (u.insertnew x ;
            escape λout. loop u.pick out λy. (g y).iter λz.
                if ∼t.member z then u.insertnew z else skip ;
            p t)
```

Notice that we have retained the *iter* field for objects of type **sett**, even though this procedure is never used in our program. The reason is that the result of *reachable* is an object of type **sett**, for which the user of *reachable* may need an iteration procedure.

Now we define the representation of t and u by programming *newdoubleset*. Within this declarator, we represent t by a characteristic vector c, which is a boolean variable array that is indexed by nodes, i.e. a procedure of type **node** → **boolvar**, such that

$$t = \{\, y \mid y\colon \mathbf{node} \wedge c\,y = \mathit{true} \,\}\,.$$

We also represent both t and u by a node variable sequence w that (with the help of two integer variables a and b) enumerates the members of these sets without duplication. Specifically,

$$\begin{aligned} t &= \{\, w\,k \mid 0 \leq k < b \,\}\,, \\ u &= \{\, w\,k \mid a \leq k < b \,\}\,. \end{aligned}$$

Thus we have

$$\begin{array}{l}
\mathbf{letinline}\ \mathit{newdoubleset}\colon (\mathbf{sett} \rightarrow \mathbf{setu} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm} \equiv \lambda p. \\
\quad \mathit{newboolvarnodearray}(\lambda n.\, \mathit{false})\ \lambda c.\ \mathit{newnodevarseq}\ N\ (\lambda k.\ \mathit{dummynode})\ \lambda w. \\
\quad \mathit{newintvar}\ 0\ \lambda a.\ \mathit{newintvar}\ 0\ \lambda b. \\
\qquad p\Big(\mathit{member} \equiv c, \\
\qquad\qquad \mathit{iter} \equiv \lambda d.\, \mathit{for}\ 0\ (b-1)\ \lambda k.\ d(w\,k)\Big) \\
\qquad \Big(\mathit{insertnew} \equiv \lambda n.\ (c\,n := \mathit{true}\ ;\ w\,b := n\ ;\ b := b+1), \\
\qquad\qquad \mathit{pick} \equiv \lambda e.\ \lambda d.\ \mathbf{if}\ a \geq b\ \mathbf{then}\ e\ \mathbf{else}\ (d(w\,a)\ ;\ a := a+1)\Big)
\end{array}$$

Here N is an integer expression giving an upper bound on the number of nodes, and *dummynode* is an arbitrary entity of type **node** used to give a spurious initialization to w.

Next, we consider the representation of graphs. As far as *reachable* is concerned, a graph is simply its immediate-successor function, of type **node** → **setg**. But the part of the program that creates graphs must have some primitive procedure for graph construction. Thus we make **graph** an object type with a field named *addedge*, denoting a procedure that, given its source and destination nodes, adds an edge to the graph:

$$\mathbf{lettype\ graph} \equiv (\mathbf{node} \rightarrow \mathbf{setg}\ \&\ \mathit{addedge}\colon \mathbf{node} \rightarrow \mathbf{node} \rightarrow \mathbf{comm})$$

Notice that the immediate-successor function is a "nameless" field of a graph, so that a graph can be passed directly to *reachable*.

We choose to represent a graph by an integer variable array *succlist*, indexed by nodes, such that *succlist* n is a list of the immediate successors of n. The lists are represented by a node variable sequence *car* and an integer variable sequence *cdr*. The integer variable k gives the number of active list cells. The empty list is represented by -1.

Thus the declarator for graphs is:

```
$\textbf{letinline}\ newgraph\colon (\textbf{graph} \to \textbf{comm}) \to \textbf{comm} \equiv \lambda p.$
    $newintvarnodearray(\lambda n.\ -1)\ \lambda succlist.$
    $newnodevarseq\ E\ (\lambda k.\ dummynode)\ \lambda car.$
    $newintvarseq\ E\ (\lambda k.\ -1)\ \lambda cdr.\ newintvar\ 0\ \lambda k.$
    $p\big(\lambda n.\ \big(iter \equiv \lambda d.\ newintvar(succlist\ n)\ \lambda l.$
            $\textbf{while}\ l \neq -1\ \textbf{do}\ (d(car\ l)\,;\ l := cdr\ l)\big),$
        $addedge \equiv \lambda m.\ \lambda n.$
            $(car\ k := n\,;\ cdr\ k := succlist\ m\,;\ succlist\ m := k\,;\ k := k+1)\big)$
```

Here E is an upper bound on the number of edges in the graph.

Next, we consider extending our program so that, in addition to determining the set of nodes that can be reached from x, it computes paths from x to each of these nodes. We will alter *reachable* so that it gives its parameter p an additional argument r of type **paths**, where an object of type **paths** provides two procedures for iterating over paths in forward and backward directions:

$$\textbf{lettype paths} \equiv \big(forward,\ backward\colon \textbf{node} \to (\textbf{node} \to \textbf{comm}) \to \textbf{comm}\big)$$

If r is an object of type **paths**, y is a node reachable from x, and d is a procedure of type **node** → **comm**, then $r.forward\ \ y\ d$ or $r.backward\ \ y\ d$ will apply d to each node on the path from x to y.

Within *reachable*, each time an immediate successor z of y is inserted in t and u, the path to z formed by adding z to the already known path to y will be recorded in r. Thus, within *reachable*, r will have the type

$$\textbf{lettype pathsvar} \equiv (\textbf{paths}\ \&\ record\colon \textbf{node} \to \textbf{node} \to \textbf{comm})$$

where $r.\,record$ is a procedure such that $r.\,record\ \ z\ y$ records the path to z formed by adding z to the path to y. (In choosing the name **pathsvar** we are stretching the meaning of "**var**". Although an object of type **pathsvar** cannot be assigned to, in the conventional sense, it still consists of an object of type **paths** intersected with an operation that changes the state of the object.)

The new version of *reachable* is:

```
$\textbf{let}\ reachable\colon \textbf{node} \to (\textbf{node} \to \textbf{setg}) \to (\textbf{sett} \to \textbf{paths} \to \textbf{comm}) \to \textbf{comm} \equiv$
    $\lambda x.\ \lambda g.\ \lambda p.\ newdoubleset\ \lambda t.\ \lambda u.\ newpathsvar\ x\ \lambda r.$
        $\big(u.insertnew\ x\,;$
        $escape\ \lambda out.\ \textbf{loop}\ u.pick\ out\ \lambda y.\ (g\,y).iter\ \lambda z.$
            $\textbf{if}\ \sim t.member\ z\ \textbf{then}\ u.insertnew\ z\,;\ r.record\ z\ y\ \textbf{else}\ skip\,;$
        $p\,t\,r\big)$
```

The representation of paths is defined within the declarator *newpathsvar*. A node variable array *link*, indexed by nodes, is used to record the calls of *record*, so that *link* $z = y$ holds after a call of r. *record* z y. Then *forward* scans *link* recursively, while *backward* scans *link* iteratively:

```
letinline newpathsvar: node → (pathsvar → comm) → comm ≡ λx. λp.
    newnodevarnodearray(λn. dummynode) λlink.
    p(record ≡ λz. λy. link z := y,
        forward ≡ λn. λd.
            letrec scan: node → comm
            where scan ≡ λn. newnodevar n λn.
                if eqnode n x then d x else (scan(link n) ; d n)
            in scan n,
        backward ≡ λn. λd. newnodevar n λn.
            (while ∼ eqnode n x do (d n ; n := link n) ; d x))
```

Here *eqnode* is a primitive operation for comparing nodes. The initial parameter x of *newpathsvar* is the node from which all the paths emanate.

Finally, we must define the data type **node**. Forsythe lacks facilities for defining new data types, but the effect of a new data type can be obtained by defining the relevant phrase types, primitive operations, and declarators. This is easy if we use a trivial representation, where a node is represented by an integer n such that $0 \leq n < N$:

```
lettype node ≡ int,
    nodeacc ≡ intacc,
    nodevar ≡ intvar
in
letinline dummynode ≡ −1,
    eqnode ≡ λm: node. λn: node. m = n,
    newnodevar ≡ newintvar,
    newnodevarseq ≡ newintvarseq,
    newboolvarnodearray ≡ newboolvarseq N,
    newintvarnodearray ≡ newintvarseq N,
    newnodevarnodearray ≡ newintvarseq N
```

Unfortunately, this way of defining **node** is limited by the fact that **lettype** definitions are transparent rather than opaque. Thus typechecking would not detect an erroneous operation that treated nodes as integers.

To avoid this difficulty, one would like to have opaque type definitions in Forsythe. However, even in the absence of opaque definitions, one can still achieve a degree of data abstraction by defining nodes to be one-field objects containing integers, rather than "raw" integers. This approach assures that the integrity of the abstraction **node** will not be violated by the reachability program providing this program does not contain any occurrence of the field name used in the definition of **node**.

This approach is embodied in the following definitions, in which *nn* is used as the "secret" field name:

```
lettype node ≡ (nn: int) in
lettype nodeacc ≡ node → comm in
lettype nodevar ≡ (node & nodeacc) in
lettype nodevarseq ≡ (int → nodevar & len: int),
    boolvarnodearray ≡ node → boolvar,
    intvarnodearray ≡ node → intvar,
    nodevarnodearray ≡ node → nodevar
in
letinline dummynode ≡ (nn ≡ −1),
    eqnode: node → node → bool ≡ λm. λn. m.nn = n.nn,
    newnodevar: node → (nodevar → comm) → comm ≡
        λinit. λb. newintvar (init.nn) λx. b(nn ≡ x, λm. x := m.nn),
    newnodevarseq: int → (int → node) → (nodevarseq → comm) → comm ≡
        λl. λinit. λb. newintvarseq l (λk. (init k).nn) λx.
            b(λk. (nn ≡ x k, λm. x k := m.nn), len ≡ x.len)
in
letinline newboolvarnodearray: (node → bool) →
            (boolvarnodearray → comm) → comm ≡
        λinit. λb. newboolvarseq N (λk. init(nn ≡ k)) λx. b(λn. x(n.nn)),
    newintvarnodearray: (node → int) →
            (intvarnodearray → comm) → comm ≡
        λinit. λb. newintvarseq N (λk. init(nn ≡ k)) λx. b(λn. x(n.nn)),
    newnodevarnodearray: (node → node) →
            (nodevarnodearray → comm) → comm ≡
        λinit. λb. newnodevarseq N (λk. init(nn ≡ k)) λx. b(λn. x(n.nn))
```

This approach to defining a new data type by defining the relevant phrase types can be used to provide more interesting representations. As a final example, we define complex numbers:

$$
\begin{array}{l}
\mathbf{lettype}\ \mathit{complex} \equiv (r\colon \mathbf{real}\ \&\ i\colon \mathbf{real})\ \mathbf{in} \\
\mathbf{lettype}\ \mathit{complexacc} \equiv \mathit{complex} \to \mathbf{comm}\ \mathbf{in} \\
\mathbf{lettype}\ \mathit{complexvar} \equiv (\mathit{complex}\ \&\ \mathit{complexacc})\ \mathbf{in} \\
\mathbf{letinline}\ \mathit{newcomplexvar}\colon \mathit{complex} \to (\mathit{complexvar} \to \mathit{comm}) \to \mathit{comm} \equiv \\
\quad \lambda \mathit{init}.\ \lambda \mathit{body}.\ \mathit{newrealvar}\ \mathit{init}.r\ \lambda \mathit{rpart}.\ \mathit{newrealvar}\ \mathit{init}.i\ \lambda \mathit{ipart}. \\
\qquad \mathit{body}\Big(r \equiv \mathit{rpart}, i \equiv \mathit{ipart}, \\
\qquad\qquad \lambda c.\ \mathit{newrealvar}\ c.r\ \lambda t.\ (\mathit{ipart} := c.i\,;\ \mathit{rpart} := t)\Big)\,.
\end{array}
$$

Here *newcomplexvar* is a declarator that represents a complex variable by two real variables *rpart* and *ipart.* The last line defines an acceptor that sets the representation variables. (Note the use of the temporary t to insure that both parts of the complex argument c are evaluated before either representation variable is set.)

Of course, one would like to make real numbers a subtype of complex numbers, with an implicit conversion that sets *ipart* to zero. In Forsythe, however, this is not permitted. Unfortunately, there seems to be no way to allow such user-defined implicit conversions while enforcing the relationships between conversions and procedures with multiple types that are described in Section 4.

14. Other Publications Related to Forsythe

The genesis of Forsythe lies in the author's general viewpoint about Algol-like languages [5], and especially in the functor-category semantics of such languages developed by F. Oles [16, 17]. The language was first described in a preliminary report [1], of which this document is a substantial revision. (The most important change in the language is that the requirements for explicit type information have been made more flexible.)

There are several technical problems associated with intersection types. The semantics of intersection as a pullback is not syntax-directed, since the meaning of $\theta_1\ \&\ \theta_2$ depends upon then meaning of $\theta_1 \sqcup \theta_2$ as well as of θ_1 and of θ_2. A demonstration that the semantics is still well-defined was given in an invited talk at the Logic in Computer Science Symposium [19], but was never written up. A proof that the semantics is coherent, i.e. that different proofs of the same typing do not lead to different meanings, was given in [20].

The functor-category semantics is the basis of a scheme for generating intermediate code for Algol-like language [21].

15. Conclusions and Future Research

There are a number of directions in which it would be desirable to extend Forsythe, providing such extensions do not impact the uniformity of the language. We are currently investigating, or hope to investigate, the following possibilities:

- Sums or disjunctions of phrase types. Unfortunately, sums of phrase types interact with the conditional construction in a counterintuitive manner. Suppose, for example, that $\mathbf{comm} + \mathbf{int}$ is the binary sum of **comm** and **int**, with injection operations in_1 of type $\mathbf{comm} \rightarrow (\mathbf{comm} + \mathbf{int})$ and in_2 of type $\mathbf{int} \rightarrow (\mathbf{comm} + \mathbf{int})$, and a case operation $\oplus$ such that, for any phrase type θ, if p_1 has type $\mathbf{comm} \rightarrow \theta$ and p_2 has type $\mathbf{int} \rightarrow \theta$ then $p_1 \oplus p_2$ has type $(\mathbf{comm} + \mathbf{int}) \rightarrow \theta$, with the reduction rule

  $$(p_1 \oplus p_2)(in_i\ x) \quad \Longrightarrow \quad p_i\, x\,.$$

  If application of the conditional construction to sum types is to make sense, then the reduction

  $$(p_1 \oplus p_2)(\mathbf{if}\, b\, \mathbf{then}\, in_1\ c\, \mathbf{else}\, in_2\ e) \quad \Longrightarrow \quad \mathbf{if}\, b\, \mathbf{then}\, p_1\ c\, \mathbf{else}\, p_2\ e$$

  must hold. Then, if the language is to exhibit reasonably uniform behavior, the similar reduction

  $$(p_1 \oplus p_2)(\mathbf{if}\, b\, \mathbf{then}\, in_1\ c\, \mathbf{else}\, in_1\ c') \quad \Longrightarrow \quad \mathbf{if}\, b\, \mathbf{then}\, p_1\ c\, \mathbf{else}\, p_1\ c'$$

  must also hold. But then, the reduction

  $$in_1(\mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, c') \quad \Longrightarrow \quad \mathbf{if}\, b\, \mathbf{then}\, in_1\ c\, \mathbf{else}\, in_1\ c'$$

  cannot hold, for otherwise we would have both

  $$(p_1 \oplus p_2)\Big(in_1(\mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, c')\Big) \quad \Longrightarrow \quad p_1(\mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, c')$$

  and

  $$\begin{aligned}(p_1 \oplus p_2)\Big(in_1(\mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, c')\Big) \quad &\Longrightarrow \quad (p_1 \oplus p_2)(\mathbf{if}\, b\, \mathbf{then}\, in_1\ c\, \mathbf{else}\, in_1\ c') \\ &\Longrightarrow \quad \mathbf{if}\, b\, \mathbf{then}\, p_1\ c\, \mathbf{else}\, p_1\ c'\,,\end{aligned}$$

  which reduce the same phrase to two phrases that will have different meanings if the procedure p_1 changes the value of b before executing its parameter. In particular, the falsity of the reduction rule for injections and conditionals implies that injections cannot be treated as implicit conversions.

- Polymorphic or universally quantified phrase types [22], possibly with bounded quantification in the sense of [23]. In addition to providing polymorphic procedures, this extension would also provide opaque type definitions. This kind of extension has been investigated by B. C. Pierce [18, 24], who found that there is no decision procedure for type-checking bounded quantification, even in the absence of intersection types. On the other hand, Pierce gave a practical algorithm for type-checking the combination of bounded quantification and intersection types that seems to terminate in cases of practical interest. He also gave a number of interesting examples of the descriptive power of such a type system.

- Recursively defined phrase types.
- Enriched data types. Although the data types of Algol (and so far of Forsythe) are limited to primitive, unstructured types, there would be no inconsistency in providing a much richer variety of data types. The real question is which of the many possible enrichments would provide additional expressive power without degrading efficiency of execution.
- Coroutines.
- Alternative treatment of arrays. Array facilities along the lines of those described in [25] would serve to avoid spurious array initializations, such as the initialization of w in *newdoubleset* in Section 13. But it is not clear how this approach can be extended to encompass multidimensional arrays.

On the other hand, there is also a direction in which it might be fruitful to restrict Forsythe: to impose syntactic restrictions so that one can determine syntactically (in a fail-safe manner) when phrases cannot interfere with one another. (Two phrases *interfere* if their concurrent execution is indeterminate. For example, aliased variables interfere, as do procedures that assign to the same global variables.) Such a restriction would open the door for the concurrent, yet determinate, execution of noninterfering commands, as well as for a form of block expression (in the sense of Algol W) that is restricted to avoid side effects.

Nearly two decades ago, I wrote a paper [26] proposing a scheme for restricting Algol-like languages for this purpose. At the time, certain syntactic anomalies (described in the final section of [26]) discouraged me from pursuing the matter further. But it is now clear that these anomalies can be avoided [27, 28]. Moreover, it appears that this approach does not raise insuperable type-checking complications.

However, the syntactic discipline described in these papers would still restrict Forsythe uncomfortably. For example, one could not regard $a := e$ as $a\ e$ when a and e interfere, nor could one write *newintvar init b* when *init* and b interfere. There are also problems with recursive procedures that assign to global variables, and the use of completions is precluded.

References

[1] Reynolds, J. C. *Preliminary Design of the Programming Language Forsythe.* Report, no. CMU–CS–88–159, Carnegie Mellon University, Computer Science Department, June 21, 1988.

[2] Naur, P. et al. *Report on the Algorithmic Language ALGOL 60.* **Communications of the ACM**, vol. 3 (1960), pp. 299–314.

[3] Naur, P. et al. *Revised Report on the Algorithmic Language ALGOL 60.* **Communications of the ACM**, vol. 6 (1963), pp. 1–17.

[4] Wirth, N. and Hoare, C. A. R. *A Contribution to the Development of ALGOL.* **Communications of the ACM**, vol. 9 (1966), pp. 413–432.

[5] Reynolds, J. C. *The Essence of Algol.* in: **Algorithmic Languages**, Proceedings of the International Symposium on Algorithmic Languages, Amsterdam, October 26–29, edited by J. W. de Bakker and J. C. van Vliet. North-Holland, Amsterdam, 1981, pp. 345–372.

[6] Landin, P. J. *The Next 700 Programming Languages.* **Communications of the ACM**, vol. 9 (1966), pp. 157–166.

[7] van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., and Fisker, R. G. *Revised Report on the Algorithmic Language ALGOL 68.* **Acta Informatica**, vol. 5 (1975), pp. 1–236.

[8] Sussman, G. J. and Steele Jr., G. L. *SCHEME: An Interpreter for Extended Lambda Calculus.* AI Memo, no. 349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, December 1975, 42 pp.

[9] Milner, R., Tofte, M., and Harper, R. W. **The Definition of Standard ML**. MIT Press, Cambridge, Massachusetts, 1990, xi+101 pp.

[10] Reynolds, J. C. **The Craft of Programming**. Prentice-Hall International, London, 1981, xiii+434 pp.

[11] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. *SIMULA 67 Common Base Language.* Publication, no. S–2, Norwegian Computing Center, Oslo, Norway, May 1968.

[12] Cardelli, L. *A Semantics of Multiple Inheritance.* in: **Semantics of Data Types**, International Symposium, Sophia-Antipolis, France, June 27–29, edited by G. Kahn, D. B. MacQueen, and G. D. Plotkin. **Lecture Notes in Computer Science**, vol. 173, Springer-Verlag, Berlin, 1984, pp. 51–67.

[13] Coppo, M., Dezani-Ciancaglini, M., and Venneri, B. *Functional Characters of Solvable Terms.* **Zeitschrift für Mathematische Logik und Grundlagen der Mathematik**, vol. 27 (1981), pp. 45–58.

[14] Coppo, M., Dezani-Ciancaglini, M., Honsell, F., and Longo, G. *Extended Type Structures and Filter Lambda Models.* in: **Logic Colloquium '82**, Florence, Italy, August 23–28, 1982, edited by G. Lolli, G. Longo, and A. Marcja. **Studies in Logic and the Foundations of Mathematics**, vol. 112, North-Holland, Amsterdam, 1984, pp. 241–262.

[15] Hindley, J. R. *Types with Intersection: An Introduction.* **Formal Aspects of Computing**, vol. 4 (1992), pp. 470–486.

[16] Oles, F. J. *A Category-Theoretic Approach to the Semantics of Programming Languages*, Ph. D. Dissertation. Syracuse University, August 1982, vi+240 pp.

[17] Oles, F. J. *Type Algebras, Functor Categories, and Block Structure.* in: **Algebraic Methods in Semantics**, edited by M. Nivat and J. C. Reynolds. Cambridge University Press, Cambridge, England, 1985, pp. 543–573.

[18] Pierce, B. C. *Programming with Intersection Types and Bounded Polymorphism*, Ph. D. Dissertation. Carnegie Mellon University, December 1991, viii+175 pp. *Report No. CMU-CS-91-205.*

[19] Reynolds, J. C. *Conjunctive Types and Algol-like Languages (Abstract of Invited Lecture).* in: **Proceedings Symposium on Logic in Computer Science**, Ithaca, New York, June 22–25. 1987, p. 119.

[20] Reynolds, J. C. *The Coherence of Languages with Intersection Types.* in: **Theoretical Aspects of Computer Software**, International Conference TACS '91, Proceedings, Sendai, Japan, September 24–27, 1991, edited by T. Ito and A. R. Meyer. **Lecture Notes in Computer Science**, vol. 526, Springer-Verlag, Berlin, 1991, pp. 675–700.

[21] Reynolds, J. C. *Using Functor Categories to Generate Intermediate Code.* in: **Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**, San Francisco, January 22–25. 1995, pp. 25–36.

[22] Reynolds, J. C. *Towards a Theory of Type Structure.* in: **Programming Symposium**, Proceedings, Colloque sur la Programmation, Paris, April 9–11, edited by B. Robinet. **Lecture Notes in Computer Science**, vol. 19, Springer-Verlag, Berlin, 1974, pp. 408–425.

[23] Cardelli, L. and Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* **ACM Computing Surveys**, vol. 17 (1985), pp. 471–522.

[24] Pierce, B. C. *Bounded Quantification is Undecidable.* in: **Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**, Albuquerque, New Mexico, January 19–22. 1992, pp. 305–315.

[25] Dijkstra, E. W. **A Discipline of Programming**. Prentice-Hall, Englewood Cliffs, New Jersey, 1976, xviii+217 pp.

[26] Reynolds, J. C. *Syntactic Control of Interference.* in: **Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages**, Tucson, Arizona, January 23–25. 1978, pp. 39–46.

[27] Reynolds, J. C. *Syntactic Control of Interference, Part 2.* in: **Automata, Languages and Programming**, 16th International Colloquium, Stresa, Italy, July 11–15, edited by G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca. **Lecture Notes in Computer Science**, vol. 372, Springer-Verlag, Berlin, 1989, pp. 704–722.

[28] O'Hearn, P. W., Power, A. J., Takeyama, M., and Tennent, R. D. *Syntactic Control of Interference Revisited.* in: **Mathematical Foundations of Programming Semantics, Eleventh Annual Conference**, edited by S. Brookes, M. Main, A. Melton, and M. Mislove. **Electronic Notes in Theoretical Computer Science**, vol. 1, Elsevier Science (http://www.elsevier.nl), Tulane University, New Orleans, Louisiana, 1995.

[29] Hopcroft, J. E. and Ullman, J. D. **Introduction to Automata Theory, Languages, and Computation**. Addison-Wesley, Reading, Massachusetts, 1979, x+418 pp.

APPENDICES

A Lexical Structure

A Forsythe program, in the form actually read by a computer, is an ASCII character string that is interpreted as a sequence of *lexemes* and *separators*. (After eliminating the separators, the sequence of lexemes is interpreted according to the concrete syntax given in Appendix B.)

A lexeme is one of the following:

- A keyword, which is one of the following character strings:

```
and     begin    div          do        else
end     error    if           iff       implies
in      let      letinline    letrec    lettype
loop    or       rec          rem       seq
then    where    while
```

- An identifier, i.e. ⟨id⟩, which is a sequence of one or more letters, digits, and underscore symbols that begins with a letter and is not a keyword.

  In this report, keywords, and also identifiers that are used as type identifiers, are typeset in boldface, but no such font distinction is made in the language actually read by the computer.

- A natural-number constant, i.e. ⟨nat const⟩, which is a sequence of one or more digits.

- A real-number constant, i.e. ⟨real const⟩, which is a sequence of digits and decimal points begining with a digit and containing exactly one decimal point; this sequence may optionally be followed by a scale factor, which consists of the letter `E` or `e`, an optional `+` or `-` sign, and a natural-number constant.

- A character constant, i.e. ⟨char const⟩, which is the character `#`, followed by a character item, which is one of the following:
  - a character other than the backslash `\`, the newline symbol, or the tab symbol,
  - `\\`, denoting the backslash,
  - `\n`, denoting a newline symbol,
  - `\t`, denoting a tab symbol,
  - `\'`, denoting the quotation mark `'`,
  - a backslash, followed by three digits (which are interpreted as an ASCII code in octal representation).

- A string constant, i.e. ⟨string const⟩, which is a sequence of zero or more string items, enclosed in the single quotation marks ‘ and ’, where a string item is one of the following:
  - a character other than the backslash `\`, the newline symbol, the tab symbol, or the quotation mark `'`,
  - `\\`, denoting the backslash,
  - `\n`, denoting a newline symbol,
  - `\t`, denoting a tab symbol,
  - `\'`, denoting the quotation mark `'`,
  - a backslash, followed by three digits (which are interpreted as an ASCII code in octal representation),
  - the backslash, followed by a blank, tab, or newline, followed by a sequence of zero or more characters other than a backslash, followed by a backslash.

  The last form of string item has no effect on the meaning of the string constant. It is included to allow such constants to run over more than one line of a Forsythe program.

- A special symbol, which is one of the following characters or strings:

```
,     (     )     :     &
|     \     .     ^     *
/     +     -     =     <
>     ~     ;     ->    ==
**    ~=    <=    >=    :=
```

A separator is either a blank, a tab, a newline, or a comment, where a comment is either:

- a sequence of characters enclosed in the braces { and }. If the braces { and } occur within the sequence of characters, they must be balanced.

- a percent sign % followed by a sequence of characters not containing a newline, followed by a newline.

Except that they separate lexemes that would otherwise combine into a single lexeme (when, for example, an identifier is followed by another identifier or a natural-number constant), separators have no effect on the meaning or translation of a program. More precisely, the program is interpreted by scanning the input characters from left to right, repeatedly removing the longest string that is a lexeme or separator, and then eliminating the separators from the resulting sequence.

A number of the symbols used in this report (including the concrete syntax given in Appendix B) are not available in ASCII. They must be translated into lexemes as follows:

publication	ascii	publication	ascii
→	`->`	≤	`<=`
≡	`==`	≥	`>=`
λ	`\`	∼	`~`
↑	`^`	∧	`and`
×	`*`	∨	`or`
÷	`div`	⇒	`implies`
≠	`~=`	⇔	`iff`

B Concrete Syntax

We will specify the concrete syntax of Forsythe by giving a context-free grammar that defines "parsable" programs as sequences of lexemes. (The conversion of a string of input characters into a sequence of lexemes, and also the transliteration of lexemes needed to fit the constraint of the ASCII character set, are defined in Appendix A.) Notice that a parsable program is not necessarily typable; typing is specified by the inference rules given previously and is independent of the concrete syntax.

The one novelty of this syntax is its treatment of "heavy prefixes" such as the conditional phrase. Such phrases are permitted to follow operators even when those operators have high precedence. For example one can write

$$A \times \mathbf{if}\, B \,\mathbf{then}\, C \,\mathbf{else}\, D + E$$

instead of

$$A \times (\mathbf{if}\, B \,\mathbf{then}\, C \,\mathbf{else}\, D + E)\,.$$

To illustrate the treatment of heavy prefixes, consider augmenting a simple language of arithmetic expressions,

⟨factor⟩ ::= ⟨id⟩ | (⟨expression⟩)
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ × ⟨factor⟩
⟨expression⟩ ::= ⟨term⟩ | ⟨term⟩ + ⟨expression⟩

with a conditional expression, treated as a heavy prefix. The resulting grammar is:

⟨factor⟩ ::= ⟨id⟩ | (⟨general expression⟩)
⟨heavy factor⟩ ::= **if**⟨general expression⟩ **then**⟨general expression⟩ **else**
        ⟨general expression⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ × ⟨factor⟩
⟨heavy term⟩ ::= ⟨heavy factor⟩ | ⟨term⟩ × ⟨heavy factor⟩
⟨expression⟩ ::= ⟨term⟩ | ⟨term⟩ + ⟨expression⟩
⟨heavy expression⟩ ::= ⟨heavy term⟩ | ⟨term⟩ + ⟨heavy expression⟩
⟨general expression⟩ ::= ⟨expression⟩ | ⟨heavy expression⟩

In this simple example, a phrase beginning with a heavy prefix will extend to the next right parenthesis (or to the end of the text). In the actual syntax of Forsythe, such phrases extend to the next semicolon, comma, right parenthesis, or **end**.

The grammar of Forsythe is given by the following productions, in which we use ⟨type n⟩ to denote type expressions, ⟨p n⟩ to denote phrases, and ⟨hp n⟩ to denote heavy phrases. The integer n indicates the precedence level (with small n for high precedence). The symbols

⟨id⟩, ⟨nat const⟩, ⟨real const⟩, ⟨char const⟩, and ⟨string const⟩ denote the lexeme classes for identifiers and various kinds of constants, as defined in Appendix A.

⟨program⟩ ::= ⟨p 16⟩

⟨type id⟩ ::= ⟨id⟩
⟨id list⟩ ::= ⟨id⟩ | ⟨id⟩, ⟨id list⟩

⟨type 0⟩ ::= ⟨type id⟩ | (⟨type 3⟩) | **begin**⟨type 3⟩ **end**
⟨type 1⟩ ::= ⟨type 0⟩ | ⟨type 0⟩ → ⟨type 1⟩
⟨type 2⟩ ::= ⟨type 1⟩ | ⟨id list⟩ : ⟨type 2⟩
⟨type 3⟩ ::= ⟨type 2⟩ | ⟨type 2⟩ & ⟨type 3⟩
⟨alt type⟩ ::= ⟨type 1⟩ | ⟨type 1⟩ | ⟨alt type⟩

⟨let list⟩ ::= ⟨id⟩ ≡ ⟨p 15⟩ | ⟨id⟩ : ⟨type 1⟩ ≡ ⟨p 15⟩
  | ⟨id⟩ ≡ ⟨p 15⟩, ⟨let list⟩ | ⟨id⟩ : ⟨type 1⟩ ≡ ⟨p 15⟩, ⟨let list⟩
⟨letrec list⟩ ::= ⟨id list⟩ : ⟨type 1⟩ | ⟨id list⟩ : ⟨type 1⟩, ⟨letrec list⟩
⟨where list⟩ ::= ⟨id⟩ ≡ ⟨p 15⟩ | ⟨id⟩ ≡ ⟨p 15⟩, ⟨where list⟩
⟨lettype list⟩ ::= ⟨type id⟩ ≡ ⟨alt type⟩ | ⟨type id⟩ ≡ ⟨alt type⟩, ⟨lettype list⟩
⟨seq list⟩ ::= ⟨p 15⟩ | ⟨p 15⟩, ⟨seq list⟩

⟨p 0⟩ ::= ⟨id⟩ | ⟨nat const⟩ | ⟨real const⟩ | ⟨char const⟩ | ⟨string const⟩
  | (⟨p 16⟩) | **begin**⟨p 16⟩ **end**
⟨hp 0⟩ ::= **if**⟨p 16⟩ **then**⟨p 16⟩ **else**⟨p 13⟩
  | **while**⟨p 16⟩ **do**⟨p 13⟩
  | **loop**⟨p 13⟩
  | λ⟨id list⟩ : ⟨alt type⟩. ⟨p 13⟩
  | λ⟨id list⟩. ⟨p 13⟩
  | **rec** : ⟨type 1⟩. ⟨p 13⟩
  | **let**⟨let list⟩ **in**⟨p 13⟩ | **letinline**⟨let list⟩ **in**⟨p 13⟩
  | **letrec**⟨letrec list⟩ **where**⟨where list⟩ **in**⟨p 13⟩
  | **lettype**⟨lettype list⟩ **in**⟨p 13⟩

⟨p 1⟩ ::= ⟨p 0⟩ | ⟨p 1⟩.⟨id⟩
⟨hp 1⟩ ::= ⟨hp 0⟩
⟨p 2⟩ ::= ⟨p 1⟩ | ⟨p 2⟩⟨p 1⟩ | **error**⟨p 1⟩
  | **seq**(⟨seq list⟩) | **seq begin**⟨seq list⟩ **end**
⟨hp 2⟩ ::= ⟨hp 1⟩ | ⟨p 2⟩⟨hp 1⟩ | **error**⟨hp 1⟩

⟨p 3⟩ ::= ⟨p 2⟩ | ⟨p 3⟩⟨exp op⟩⟨p 2⟩
⟨hp 3⟩ ::= ⟨hp 2⟩ | ⟨p 3⟩⟨exp op⟩⟨hp 2⟩
⟨exp op⟩ ::= ↑ | **
⟨p 4⟩ ::= ⟨p 3⟩ | ⟨p 4⟩⟨mult op⟩⟨p 3⟩
⟨hp 4⟩ ::= ⟨hp 3⟩ | ⟨p 4⟩⟨mult op⟩⟨hp 3⟩
⟨mult op⟩ ::= × | / | ÷ | **rem**
⟨p 5⟩ ::= ⟨p 4⟩ | ⟨add op⟩⟨p 4⟩ | ⟨p 5⟩⟨add op⟩⟨p 4⟩
⟨hp 5⟩ ::= ⟨hp 4⟩ | ⟨add op⟩⟨hp 4⟩ | ⟨p 5⟩⟨add op⟩⟨hp 4⟩
⟨add op⟩ ::= + | −

⟨p 6⟩ ::= ⟨p 5⟩ | ⟨p 6⟩⟨rel op⟩⟨p 5⟩
⟨hp 6⟩ ::= ⟨hp 5⟩ | ⟨p 6⟩⟨rel op⟩⟨hp 5⟩
⟨rel op⟩ ::= = | ≠ | ≤ | < | ≥ | >

⟨p 7⟩ ::= ⟨p 6⟩ | ~⟨p 6⟩
⟨hp 7⟩ ::= ⟨hp 6⟩ | ~⟨hp 6⟩
⟨p 8⟩ ::= ⟨p 7⟩ | ⟨p 8⟩ ∧ ⟨p 7⟩
⟨hp 8⟩ ::= ⟨hp 7⟩ | ⟨p 8⟩ ∧ ⟨hp 7⟩
⟨p 9⟩ ::= ⟨p 8⟩ | ⟨p 9⟩ ∨ ⟨p 8⟩
⟨hp 9⟩ ::= ⟨hp 8⟩ | ⟨p 9⟩ ∨ ⟨hp 8⟩
⟨p 10⟩ ::= ⟨p 9⟩ | ⟨p 10⟩ ⇒ ⟨p 9⟩
⟨hp 10⟩ ::= ⟨hp 9⟩ | ⟨p 10⟩ ⇒ ⟨hp 9⟩
⟨p 11⟩ ::= ⟨p 10⟩ | ⟨p 11⟩ ⇔ ⟨p 10⟩
⟨hp 11⟩ ::= ⟨hp 10⟩ | ⟨p 11⟩ ⇔ ⟨hp 10⟩
⟨p 12⟩ ::= ⟨p 11⟩ | ⟨p 12⟩ := ⟨p 11⟩
⟨hp 12⟩ ::= ⟨hp 11⟩ | ⟨p 12⟩ := ⟨hp 11⟩
⟨p 13⟩ ::= ⟨p 12⟩ | ⟨hp 12⟩

⟨p 14⟩ ::= ⟨p 13⟩ | ⟨p 14⟩ ; ⟨p 13⟩
⟨p 15⟩ ::= ⟨p 14⟩ | ⟨id⟩ ≡ ⟨p 15⟩
⟨p 16⟩ ::= ⟨p 15⟩ | ⟨p 16⟩, ⟨id⟩ ≡ ⟨p 15⟩
    | ⟨p 16⟩, λ⟨id list⟩ : ⟨alt type⟩. ⟨p 13⟩
    | ⟨p 16⟩, λ⟨id list⟩. ⟨p 13⟩

C Type Checking

It is well-known that there is no algorithm for the complete inference of intersection types [13, 14, 15]. Thus, Forsythe must require the programmer to provide a degree of explicit type information. We have attempted to make this requirement as flexible as possible; as a consequence, a precise description of where explicit types must occur is rather complicated.

The following is a grammatical schema (i.e. a van Wijngaarden grammar) for the abstract syntax of a sublanguage of the language described earlier, such that every program in the sublanguage contains enough type information to be typechecked (even though the program may not be type-correct). The converse is nearly true, though (using phrases of type **ns**) one can contrive programs that typecheck even though they do not satisfy this schema.

The nonterminal symbols $\langle \mathrm{p}_n \rangle$ and $\langle \mathrm{seq\ list}_n \rangle$ are indexed by nonnegative integers and infinity, with $\infty \pm 1 = \infty$ and $0 - 1 = 0$. It is assumed that syntactic sugar (excepting definitional forms) has been eliminated as in Section 7, and that ⟨type⟩, ⟨alt type⟩, ⟨lettype list⟩, and ⟨letrec list⟩ are defined as in Appendix B.

$\langle \text{program} \rangle ::= \langle \mathrm{p}_0 \rangle$

$\langle \text{unary op} \rangle ::= + \mid - \mid \sim$

$\langle \text{binary op} \rangle ::= \uparrow \mid ** \mid \times \mid / \mid \div \mid \textbf{rem} \mid + \mid - \mid = \mid \neq \mid \leq \mid < \mid \geq \mid > \mid \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \mid ;$

$\langle \text{let list} \rangle ::= \langle \text{id} \rangle \equiv \langle \mathrm{p}_\infty \rangle \mid \langle \text{id} \rangle : \langle \text{type} \rangle \equiv \langle \mathrm{p}_0 \rangle$

$\quad \mid \langle \text{id} \rangle \equiv \langle \mathrm{p}_\infty \rangle, \langle \text{let list} \rangle \mid \langle \text{id} \rangle : \langle \text{type} \rangle \equiv \langle \mathrm{p}_0 \rangle, \langle \text{let list} \rangle$

$\langle \text{where list} \rangle ::= \langle \text{id} \rangle \equiv \langle \mathrm{p}_0 \rangle \mid \langle \text{id} \rangle \equiv \langle \mathrm{p}_0 \rangle, \langle \text{where list} \rangle$

$\langle \text{seq list}_n \rangle ::= \langle \mathrm{p}_{n-1} \rangle \mid \langle \mathrm{p}_{n-1} \rangle, \langle \text{seq list}_n \rangle$

$\langle \mathrm{p}_n \rangle ::= \langle \text{id} \rangle \mid \langle \text{nat const} \rangle \mid \langle \text{real const} \rangle \mid \langle \text{char const} \rangle \mid \langle \text{string const} \rangle$

$\quad \mid \textbf{if} \langle \mathrm{p}_0 \rangle \textbf{ then} \langle \mathrm{p}_n \rangle \textbf{ else} \langle \mathrm{p}_n \rangle$

$\quad \mid \textbf{while} \langle \mathrm{p}_0 \rangle \textbf{ do} \langle \mathrm{p}_0 \rangle \mid \textbf{loop} \langle \mathrm{p}_0 \rangle$

$\quad \mid \lambda \langle \text{id} \rangle : \langle \text{alt type} \rangle.\ \langle \mathrm{p}_{n-1} \rangle \mid \langle \mathrm{p}_n \rangle, \lambda \langle \text{id} \rangle : \langle \text{alt type} \rangle.\ \langle \mathrm{p}_{n-1} \rangle$

$\quad \mid \textbf{rec} : \langle \text{type} \rangle.\ \langle \mathrm{p}_0 \rangle$

$\quad \mid \textbf{let} \langle \text{let list} \rangle \textbf{ in} \langle \mathrm{p}_n \rangle \mid \textbf{letinline} \langle \text{let list} \rangle \textbf{ in} \langle \mathrm{p}_n \rangle$

$\quad \mid \textbf{letrec} \langle \text{letrec list} \rangle \textbf{ where} \langle \text{where list} \rangle \textbf{ in} \langle \mathrm{p}_n \rangle$

$\quad \mid \textbf{lettype} \langle \text{lettype list} \rangle \textbf{ in} \langle \mathrm{p}_n \rangle$

$\quad \mid \langle \mathrm{p}_n \rangle . \langle \text{id} \rangle$

$\quad \mid \langle \mathrm{p}_{n+1} \rangle \langle \mathrm{p}_0 \rangle$

$\quad \mid \textbf{seq}(\langle \text{seq list}_n \rangle)$

$\quad \mid \langle \text{unary op} \rangle \langle \mathrm{p}_0 \rangle \mid \langle \mathrm{p}_0 \rangle \langle \text{binary op} \rangle \langle \mathrm{p}_0 \rangle$

$\quad \mid \langle \text{id} \rangle \equiv \langle \mathrm{p}_n \rangle \mid \langle \mathrm{p}_n \rangle, \langle \text{id} \rangle \equiv \langle \mathrm{p}_n \rangle$

$\langle \mathrm{p}_0 \rangle ::= \lambda \langle \text{id} \rangle.\ \langle \mathrm{p}_0 \rangle \mid \langle \mathrm{p}_0 \rangle, \lambda \langle \text{id} \rangle.\ \langle \mathrm{p}_0 \rangle \mid \textbf{error} \langle \mathrm{p}_0 \rangle$

When the typechecker examines a phrase occurrence described by the nonterminal $\langle \mathrm{p}_n \rangle$, it is given a *goal* describing a set of potential *simple* types of the phrase that are relevant to the typing of the enclosing program. When $n = 0$ this set is finite; when $n = \infty$ it is the set of all simple types. Very roughly speaking, when n is nonzero and finite, it describes a set of procedural types whose first n arguments are arbitrary. The final production displayed above shows that certain phrases, especially abstractions without type information, are only permitted in contexts where the goal describes a finite set.

To make this sketchy description more precise, we first define a *simple type* to be a type with no occurrence of & except on the left of one or more arrows. More formally,

$$\omega ::= \rho \mid \theta \rightarrow \omega \mid \iota{:}\,\omega \qquad \text{(simple types)}$$

To within equivalence, every type is an intersection of simple types. To express this fact, we define the function s, which maps types into finite sets of simple types, as follows:

$$s\ \rho = \{\rho\}$$

$$s(\theta \rightarrow \theta') = \{\, \theta \rightarrow \omega \mid \omega \in s\ \theta' \,\}$$

$$s(\iota{:}\,\theta) = \{\, \iota{:}\,\omega \mid \omega \in s\ \theta \,\}$$

$$s\ \mathbf{ns} = \{\}$$

$$s\ (\theta_1 \mathbin{\&} \theta_2) = s\ \theta_1 \cup s\ \theta_2\ ,$$

and we define the function &, which maps finite sets of types into types, by

$$\&\{\} = \mathbf{ns}$$

$$\&\{\theta\} = \theta$$

$$\&\{\theta_1, \ldots, \theta_n, \theta_{n+1}\} = \theta_1 \mathbin{\&} \big(\&\{\theta_2, \ldots, \theta_{n+1}\}\big) \text{ when } n \geq 1\,.$$

(Strictly speaking, this definition only makes sense if one imposes some ordering on the types θ_1, ... , θ_{n+1}. But this ordering can be arbitrary, since & is commutative with respect to the equivalence of types.)

It is easy to see, by induction on the structure of simple types, that $s\ \omega = \{\omega\}$ for any ω. Moreover, it can be shown that, for any type θ and any set σ of simple types,

$$\&(s\ \theta) \simeq \theta \qquad \text{and} \qquad s(\&\,\sigma) = \sigma\,.$$

It can also be shown that

$$\theta \leq \theta' \text{ if and only if } (\forall \omega' \in s\ \theta')(\exists \omega \in s\ \theta)\ \omega \leq \omega'\,,$$

and that $\omega \leq \omega'$ if and only if one of the following conditions holds:

1. There are primitive types ρ and ρ' such that
$$\omega = \rho \text{ and } \omega' = \rho' \text{ and } \rho \leq_{\text{prim}} \rho' .$$

2. There are types θ_1 and θ_1' and simple types ω_2 and ω_2' such that
$$\omega = \theta_1 \rightarrow \omega_2 \text{ and } \omega' = \theta_1' \rightarrow \omega_2' \text{ and } \theta_1' \leq \theta_1 \text{ and } \omega_2 \leq \omega_2' .$$

3. There are an identifier ι and simple types ω_1 and ω_1' such that
$$\omega = \iota{:}\, \omega_1 \text{ and } \omega' = \iota{:}\, \omega_1' \text{ and } \omega_1 \leq \omega_1' .$$

These properties lead directly to an algorithm for computing the predicate $\theta \leq \theta'$.

As remarked earlier, a *goal* is an entity denoting a set of simple types. The simplest kind of goal is a type θ, which denotes the set $s\ \theta$. But we also need goals that denote certain infinite sets. Thus we define

$$\gamma ::= \theta \mid \mathsf{T} \mid \ \rhd \gamma \mid \iota{:}\, \gamma \qquad \text{(goals)}$$

and we extend the function s to map the new goals into the sets they represent:

$$s\, \mathsf{T} = \mathcal{S}$$

$$s(\ \rhd \gamma) = \{\, \theta \rightarrow \omega \mid \theta \in \mathcal{T} \text{ and } \omega \in s\ \gamma \,\}$$
$$s(\iota{:}\, \gamma) = \{\, \iota{:}\, \omega \mid \omega \in s\ \gamma \,\} ,$$

where $\mathcal{S}$ denotes the set of all simple types and $\mathcal{T}$ denotes the set of all types.

Finally, we define the typechecking function, *tc*, which maps a type assignment, phrase, and goal into a type. Within equivalence,

$$tc(\pi, p, \gamma) \simeq \&\{\, \omega \mid \omega \in s\ \gamma \text{ and } \pi \vdash p{:}\, \omega \,\} .$$

Thus $tc(\pi, p, \gamma)$ will be a greatest lower bound of the set of simple types ω that belong to the set denoted by the goal γ and also satisfy the typing $\pi \vdash p{:}\, \omega$. When $\gamma = \mathsf{T}$ there is no contextual information, corresponding to bottom-up typechecking. At the other extreme, top-down checking is also encompassed: The typing $\pi \vdash p{:}\, \theta$ is valid if and only if $tc(\pi, p, \theta) \leq \theta$. (In fact, this subtype relation will hold if and only if the equivalence $tc(\pi, p, \theta) \simeq \theta$ holds, since the opposite subtyping $\theta \leq tc(\pi, p, \theta)$ will always hold.)

Now we can give a precise description of the indexing in the abstract grammar at the beginning of this appendix: The nonterminal $\langle \mathrm{p}_\infty \rangle$ describes those occurrences of phrases that will be typechecked with a goal containing T, while the nonterminal $\langle \mathrm{p}_n \rangle$ describes those occurrences that will be typechecked with a goal that does not contain T, but contains n occurrences of $\rhd$.

It is important to realize that, even though some goals are also types, goals play a different role than types, and are therefore a different kind of entity. Specifically, the equivalence relation on types is inappropriate for goals, since the function tc does not map equivalent goals into equivalent types. For instance, $\mathbf{int} \simeq \mathbf{int} \,\&\, \mathbf{real}$, but (for any type assignment π),

$$tc(\pi, 0.5, \mathbf{int}) = \mathbf{ns} \qquad \text{and} \qquad tc(\pi, 0.5, \mathbf{int} \,\&\, \mathbf{real}) = \mathbf{real}\,,$$

which are not equivalent types.

The solution to this problem is to adopt a different equivalence relation $\cong$ for goals:

$$\gamma \cong \gamma' \text{ iff } (\forall \omega \in s\ \gamma)(\exists \omega' \in s\ \gamma')\ \omega \simeq \omega' \text{ and } (\forall \omega' \in s\ \gamma')(\exists \omega \in s\ \gamma)\ \omega \simeq \omega'\,.$$

For this relation, one can show that, if $\gamma \cong \gamma'$ then $tc(\pi, p, \gamma) \simeq tc(\pi, p, \gamma')$.

Under certain circumstances, the typechecker can require time that is exponential in the length of its input. This can happen because a single call $tc(\pi, p, \gamma)$ can cause more than one recursive call for the same subphrase of p under any of the following circumstances:

1. p is a **lettype** declaration containing an alternative type construction with several alternatives,

2. p is an explicitly typed abstraction containing an alternative type construction with several alternatives,

3. p is an implicitly typed abstraction and γ is an intersection of several procedural types.

One can expect the programmer to be aware of what is happening in the first two cases, since the multiple alternatives would occur explicitly in the program. The last case, however, can be more insideous and subtle. For instance, consider the call

$$tc(\pi, \mathbf{let}\ c \equiv \mathit{newintvar}\ \ 0\ \lambda x.\ B\ \mathbf{in} \cdots, \mathbf{comm})\,.$$

Since c is not explicitly typed, this leads to the call

$$tc(\pi, \mathit{newintvar}\ \ 0\ \lambda x.\ B, \mathsf{T})\,.$$

In turn, assuming that $\mathit{newintvar}$ 0 has the type

$$(\mathbf{intvar} \to \mathbf{comm}) \to \mathbf{comm} \,\&(\mathbf{intvar} \to \mathbf{compl}) \to \mathbf{compl} \,\&(\mathbf{intvar} \to \mathbf{int}) \to \mathbf{int} \,\& \\ (\mathbf{intvar} \to \mathbf{real}) \to \mathbf{real} \,\&(\mathbf{intvar} \to \mathbf{bool}) \to \mathbf{bool} \,\&(\mathbf{intvar} \to \mathbf{char}) \to \mathbf{char}\,,$$

this leads to the call

$$tc(\pi, \lambda x.\ B, \mathbf{intvar} \to \mathbf{comm} \,\&\, \mathbf{intvar} \to \mathbf{compl} \,\&\, \mathbf{intvar} \to \mathbf{int} \,\& \\ \mathbf{intvar} \to \mathbf{real} \,\&\, \mathbf{intvar} \to \mathbf{bool} \,\&\, \mathbf{intvar} \to \mathbf{char})\,.$$

Naively, one would expect this to leads to six calls that all typecheck the same abstraction body:

$$tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{comm}) \quad tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{compl}) \quad tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{int})$$
$$tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{real}) \quad tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{bool}) \quad tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{char})\,.$$

But in fact, the typechecker will take advantage of the equivalence $\cong$ for goals to replace the goal

$$\mathbf{intvar} \to \mathbf{comm} \,\&\, \mathbf{intvar} \to \mathbf{compl} \,\&\, \mathbf{intvar} \to \mathbf{int} \,\&$$
$$\mathbf{intvar} \to \mathbf{real} \,\&\, \mathbf{intvar} \to \mathbf{bool} \,\&\, \mathbf{intvar} \to \mathbf{char}$$

by the equivalent goal

$$\mathbf{intvar} \to (\mathbf{comm} \,\&\, \mathbf{compl} \,\&\, \mathbf{int} \,\&\, \mathbf{real} \,\&\, \mathbf{bool} \,\&\, \mathbf{char})\,,$$

which leads to the single call

$$tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{comm} \,\&\, \mathbf{compl} \,\&\, \mathbf{int} \,\&\, \mathbf{real} \,\&\, \mathbf{bool} \,\&\, \mathbf{char})\,.$$

Although a full discussion of the subject is beyond the scope of this report, this example illustrates how a careful choice of canonical forms for types and goals can enhance the efficiency of typechecking. Among the programs in this report, the only implicitly typed abstractions whose goals necessitate checking their body more than one are:

1. In Section 10, the binding of *fin* in the final definition of *newintvarres*,

2. In Section 11, the bindings of *b* in the initial definition of *newintvararray* and the definition of *newtrivararray*,

3. In Section 11, the bindings of X in the definitions of *slice* and *slicecheck*.

Further experience will be needed, however, before we can be confident that our typechecker is reasonably efficient in practice. As an illustration of how close we are to the edge of disaster, notice that we have avoided the temptation of giving the declarator *newintvar* 0 the type

$$(\mathbf{intvar} \to \mathbf{comm}) \to \mathbf{comm} \,\&(\mathbf{intvar} \to \mathbf{compl}) \to \mathbf{compl} \,\&(\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \,\&$$
$$(\mathbf{int} \to \mathbf{real}) \to \mathbf{real} \,\&(\mathbf{int} \to \mathbf{bool}) \to \mathbf{bool} \,\&(\mathbf{int} \to \mathbf{char}) \to \mathbf{char}\,,$$

which makes explicit the fact that local integer variables cannot be assigned within expressions. With this choice of type, the typechecking call

$$tc(\pi, \mathit{newintvar}\ \ 0\ \lambda x.\ B, \mathsf{T})$$

would lead to two calls for B:

$$tc([\,\pi \mid x\!: \mathbf{intvar}\,], B, \mathbf{comm} \,\&\, \mathbf{compl}) \qquad tc([\,\pi \mid x\!: \mathbf{int}\,], B, \mathbf{int} \,\&\, \mathbf{real} \,\&\, \mathbf{bool} \,\&\, \mathbf{char})\,,$$

so that typechecking would become exponential in the number of nested variable declarations.

Despite this cautionary example, we hope that our typechecker, perhaps with further refinements, will be reasonably efficient in normal practice. It should be noted, however, that worst-case inefficiency is inevitable. In fact, it can be shown that any typechecker for Forsythe (or any other language using intersection types) is PSPACE-hard.

The proof is obtained by reducing the problem of evaluating quantified Boolean expressions, which is known to be PSPACE-complete [29], to the type inference problem. The reduction is obtained by translating a quantified Boolean expression B into a Forsythe phrase B^* as follows:

$$\begin{aligned}(B_1 \wedge B_2)^* &= \mathit{And}\, B_1^*\, B_2^* \\ (B_1 \vee B_2)^* &= \mathit{Or}\, B_1^*\, B_2^* \\ (\neg B)^* &= \mathit{Not}\, B^* \\ ((\forall x)B)^* &= \mathit{Forall}\,(\lambda x : \mathbf{t} \mid \mathbf{f}.\ B^*) \\ ((\exists x)B)^* &= \mathit{Exists}\,(\lambda x : \mathbf{t} \mid \mathbf{f}.\ B^*) \\ x^* &= x\ ,\end{aligned}$$

where $\mathbf{t}$ and $\mathbf{f}$ are distinct types, neither of which is a subtype of the other, and *And*, *Or*, *Not*, *Forall*, and *Exists* are identifiers not occurring in the original expression. (For the particular typechecker described in this appendix, one can omit the alternative type expressions $\mathbf{t} \mid \mathbf{f}$.) In addition a truth value b is translated into a type b^* by

$$\mathit{true}^* = \mathbf{t} \qquad\qquad \mathit{false}^* = \mathbf{f}\ .$$

Let β be an assignment of truth values to the free variables of a quantified Boolean expression, and let π be the type assignment that maps each of these variables x into $(\beta x)^*$, and maps the additional variables of B^* as follows:

$$\begin{aligned}\pi(\mathit{And}) &= (\mathbf{f} \to \mathbf{f} \to \mathbf{f})\ \&\ (\mathbf{f} \to \mathbf{t} \to \mathbf{f})\ \&\ (\mathbf{t} \to \mathbf{f} \to \mathbf{f})\ \&\ (\mathbf{t} \to \mathbf{t} \to \mathbf{t}) \\ \pi(\mathit{Or}) &= (\mathbf{f} \to \mathbf{f} \to \mathbf{f})\ \&\ (\mathbf{f} \to \mathbf{t} \to \mathbf{t})\ \&\ (\mathbf{t} \to \mathbf{f} \to \mathbf{t})\ \&\ (\mathbf{t} \to \mathbf{t} \to \mathbf{t}) \\ \pi(\mathit{Not}) &= (\mathbf{f} \to \mathbf{t})\ \&\ (\mathbf{t} \to \mathbf{f}) \\ \pi(\mathit{Forall}) &= \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{f}\big)\ \&\ \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{f}\big) \\ &\quad \&\big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{f}\big)\ \&\ \big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{t}\big) \\ \pi(\mathit{Exists}) &= \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{f}\big)\ \&\ \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{t}\big) \\ &\quad \&\big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{t}\big)\ \&\ \big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{t}\big)\ .\end{aligned}$$

Then it is easy to see that B evaluates to b under the truth-value assignment β if and only if the typing $\pi \vdash B^* : b^*$ is valid.

One might object that this reduction maps closed quantified Boolean expressions into open (and type-open) phrases of Forsythe, and thus might not imply the inefficiency of typechecking closed phrases. This objection can be overcome, however, by enclosing B^* in the following declarations (which are based on the classical lambda-calculus encoding of boolean values by the projections $\lambda x.\ \lambda y.\ x$ and $\lambda x.\ \lambda y.\ y$):

$$
\begin{array}{l}
\mathbf{lettype}\ \mathbf{t} \equiv \mathbf{int} \to \mathbf{ns} \to \mathbf{int},\ \mathbf{f} \equiv \mathbf{ns} \to \mathbf{int} \to \mathbf{int} \\
\mathbf{in} \\
\mathbf{let}\ \mathit{And}\colon (\mathbf{f} \to \mathbf{f} \to \mathbf{f})\ \&\ (\mathbf{f} \to \mathbf{t} \to \mathbf{f})\ \&\ (\mathbf{t} \to \mathbf{f} \to \mathbf{f})\ \&\ (\mathbf{t} \to \mathbf{t} \to \mathbf{t}) \equiv \\
\qquad\qquad \lambda p.\ \lambda q.\ \lambda x.\ \lambda y.\ p\,(q\ x\ y)\ y \\
\qquad \mathit{Or}\colon (\mathbf{f} \to \mathbf{f} \to \mathbf{f})\ \&\ (\mathbf{f} \to \mathbf{t} \to \mathbf{t})\ \&\ (\mathbf{t} \to \mathbf{f} \to \mathbf{t})\ \&\ (\mathbf{t} \to \mathbf{t} \to \mathbf{t}) \equiv \\
\qquad\qquad \lambda p.\ \lambda q.\ \lambda x.\ \lambda y.\ p\ x\,(q\ x\ y) \\
\qquad \mathit{Not}\colon (\mathbf{f} \to \mathbf{t})\ \&\ (\mathbf{t} \to \mathbf{f}) \equiv \\
\qquad\qquad \lambda p.\ \lambda x.\ \lambda y.\ p\ y\ x \\
\mathbf{in} \\
\mathbf{let}\ \mathit{Forall}\colon \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{f}\big)\ \&\ \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{f}\big) \\
\qquad\qquad\quad \&\ \big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{f}\big)\ \&\ \big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{t}\big) \equiv \\
\qquad\qquad \lambda h.\ \mathit{And}\,(h\ \lambda x.\ \lambda y.\ x)\,(h\ \lambda x.\ \lambda y.\ y) \\
\qquad \mathit{Exists}\colon \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{f}\big)\ \&\ \big((\mathbf{f} \to \mathbf{f}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{t}\big) \\
\qquad\qquad\quad \&\ \big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{f}) \to \mathbf{t}\big)\ \&\ \big((\mathbf{f} \to \mathbf{t}\ \&\ \mathbf{t} \to \mathbf{t}) \to \mathbf{t}\big) \equiv \\
\qquad\qquad \lambda h.\ \mathit{Or}\,(h\ \lambda x.\ \lambda y.\ x)\,(h\ \lambda x.\ \lambda y.\ y) \\
\mathbf{in}\ \cdots .
\end{array}
$$

To obtain completely explicit typing, one can annotate the abstractions here as follows:

$$x, y\colon \mathbf{int} \mid \mathbf{ns}$$

$$p, q\colon \mathbf{t} \mid \mathbf{f}$$

$$h\colon \mathbf{f} \to \mathbf{f} \mid \mathbf{t} \to \mathbf{f} \mid \mathbf{f} \to \mathbf{t} \mid \mathbf{t} \to \mathbf{t}\ .$$

This makes it clear that our lower bound applies even to the typechecking of programs with completely explicit type information.